

R avancé

Sébastien Déjean et Thibault Laurent

2021-10-15

Contents

1	Introduction	5
1.1	Principe du document	5
1.2	Pré-requis	5
2	Manipulation de données avec R	7
2.1	Les chaînes de caractères	7
2.2	Les facteurs	20
2.3	Les dates	23
2.4	Opérations ensemblistes	26
2.5	Manipulation de bases de données	29
2.6	Tidy data	39
2.7	Gestion de données volumineuses	46
2.8	Visualiser et traiter les données manquantes	53
2.9	Répertoires et fichiers	56
2.10	Autour de l'espace de travail	58
3	Programmation	61
3.1	Quelques règles de style	61
3.2	Fixer la taille des vecteurs (si on la connaît à l'avance)	64
3.3	Temps de calcul et mémoire	65
3.4	Fonction vectorisée	68
3.5	Ecrire un code en C++	69
3.6	Insérer du code Python dans un document Markdown	71
3.7	Eviter si possible les boucles (<i>apply()</i> , <i>lapply()</i> , <i>replicate()</i> , <i>col-Sums()</i> , etc.)	72
3.8	Améliorer ses fonctions	80
3.9	A quoi servent les fonctions <i>substitute()</i> / <i>quote()</i> et <i>eval()</i> ?	86
3.10	Programmation orientée : S3/S4	92
3.11	Visualiser le code source d'une fonction	99
4	Calcul parallèle	113
4.1	Principe du calcul parallèle	113
4.2	Notion de programme maître	116
4.3	Fonction <i>set.seed()</i>	116

4.4	Syntaxe pour lancer un calcul parallèle	120
4.5	Equilibrer la répartition des tâches	129
4.6	Améliorer la répartition des tâches	132
4.7	Fonction vectorisée VS <code>calcul</code> // VS code C++	134
4.8	Reproductibilité des résultats : choix de la graine aléatoire	137
5	Visualisation de données	139
5.1	Fonctions graphiques de base VS <code>ggplot2</code>	139
5.2	Présentation de graphiques originaux	162
5.3	Présenter des tableaux de résultats	174
5.4	Graphiques interactifs	194
5.5	Mini-introduction à <code>shiny</code>	198
6	Hello bookdown	203
6.1	A section	203
7	Cross-references	205
7.1	Chapters and sub-chapters	205
7.2	Captioned figures and tables	205
8	Parts	209
9	Footnotes and citations	211
9.1	Footnotes	211
9.2	Citations	211
10	Blocks	213
10.1	Equations	213
10.2	Theorems and proofs	213
10.3	Callout blocks	213
11	Sharing your book	215
11.1	Publishing	215
11.2	404 pages	215
11.3	Metadata for sharing	215

Chapter 1

Introduction

1.1 Principe du document

Ce document fait suite aux documents suivants :

- *Pour se donner un peu d'R* disponible ici
- *Introduction à R* disponible ici

Il contient des commandes à saisir, des commentaires pour attirer l'attention sur des points particuliers et quelques questions/réponses pour tester la compréhension des notions présentées. Pour certains points particuliers, nécessitant par exemple un environnement logiciel particulier, les faits ne sont que mentionnés et il n'y a pas toujours de mise en oeuvre pratique.

1.2 Pré-requis

Même si la plupart des points abordés dans ce document ne sont pas très compliqués, ils relèvent d'une utilisation avancée de **R** et ne s'adressent donc pas au débutant en **R**. Avant d'utiliser ce document, le lecteur doit notamment savoir :

- se servir de l'aide en ligne de **R**,
- manipuler les objets de base de **R** : vecteur, matrice, liste, **data.frame**,
- programmer une fonction élémentaire.

```
bookdown::serve_book()
```


Chapter 2

Manipulation de données avec R

Packages à installer pour ce chapitre

```
install.packages(c(
  # import data:
  "foreign", "jsonlite", "readr", "readxl", "sas7bdat", "XML",
  # big data analysis
  "data.table", "ff", "ffbase",
  # matrices creuses
  "Marix",
  # character treatment
  "classInt", "glue", "stringr", "wordcloud",
  "gplots", # plotting data with ggplot2 style
  "tidyverse", "DSR", # Data Scientists toolkits
  "Amelia", "DMwR", "missForest", "naniar", # missing values treatment
  "sp", "sf", # spatial data object
  "zoo") # Time series analysis
)
```

2.1 Les chaînes de caractères

2.1.1 Fonctions de base

On considère la chaîne de caractère suivante :

```
phrase <- "Notes obtenues\nanglais: 16, Stat:14, Eco=18"
class(phrase)

## [1] "character"
```

Parmi les fonctions qui permettent de manipuler les chaînes de caractères, voici celles qui nous semblent importantes de connaître :

2.1.1.1 *nchar()*

Permet de compter le nombre de caractères de chaque élément d'un vecteur :

```
nchar(phrase)
```

```
## [1] 43
```

2.1.1.2 *substr()*

Permet d'extraire un sous-ensemble de caractères :

```
substr(phrase, start = 1, stop = 5)
```

```
## [1] "Notes"
```

2.1.1.3 *strsplit()*

Permet d'éclater une chaîne de caractères dès qu'on trouve une sous-chaîne particulière :

```
strsplit(phrase, split = " ")
```

```
## [[1]]
## [1] "Notes"           "obtenues\nanglais:" "16,"
## [4] "Stat:14,"         "Eco=18"
```

```
strsplit(phrase, split = "\n")
```

```
## [[1]]
## [1] "Notes obtenues"           "anglais: 16, Stat:14, Eco=18"
```

Remarque 1 : un espace est considéré comme une chaîne de caractère.

Remarque 2 : “\n” est un caractère spécial qui correspond à un saut de ligne (pour consulter la liste des caractères spéciaux, voir cette page web). Pour évaluer un caractère spécial dans une chaîne de caractère, on peut utiliser la fonction *cat()*:

```
cat(phrase)
```

```
## Notes obtenues
## anglais: 16, Stat:14, Eco=18
```

Remarque 3 : l'objet retourné est de type **list**. Pour convertir une **list** en un vecteur, sur lequel il est plus facile de faire de la manipulation, on utilise la fonction *unlist()*. Enfin, si on veut éclater une chaîne de caractère en fonction de plusieurs caractères, on utilise le symbole **|**. Dans l'exemple suivant, l'idée est de séparer tous les mots en présence d'un des caractères spéciaux :

```
(mots <- strsplit(phrase, split = ",|\n| |:|="))
## [[1]]
## [1] "Notes"      "obtenues"   "anglais"    ""           "16"          ""
## [7] "Stat"        "14"         ""           "Eco"        "18"
(mots <- unlist(mots))

## [1] "Notes"      "obtenues"   "anglais"    ""           "16"          ""
## [7] "Stat"        "14"         ""           "Eco"        "18"
```

Si on utilise le type **NULL** comme critère de recherche, cela a pour effet d'éclater tous les éléments de la chaîne de caractères en des caractères uniques :

```
(lettres <- strsplit(phrase, split = NULL))
## [[1]]
## [1] "N"  "o"  "t"  "e"  "s"  " "  "o"  "b"  "t"  "e"  "n"  "u"  "e"  "s"  "\n"
## [16] "a"  "n"  "g"  "l"  "a"  "i"  "s"  ":"  " "  "1"  "6"  ","  " "  "S"  "t"
## [31] "a"  "t"  ":"  "1"  "4"  ","  " "  "E"  "c"  "o"  "="  "1"  "8"
length(lettres[[1]])
## [1] 43
```

Remarque : dans le contexte d'une étude statistique, on appliquera cette fonction à des vecteurs de caractère. Par exemple :

```
strsplit(mots, split = ":")
```

```
## [[1]]
## [1] "Notes"
##
## [[2]]
## [1] "obtenues"
##
## [[3]]
## [1] "anglais"
##
## [[4]]
## character(0)
##
## [[5]]
## [1] "16"
##
## [[6]]
## character(0)
##
## [[7]]
```

```
## [1] "Stat"
##
## [[8]]
## [1] "14"
##
## [[9]]
## character(0)
##
## [[10]]
## [1] "Eco"
##
## [[11]]
## [1] "18"
```

2.1.1.4 *toupper()*, *tolower()*

Pemettent de convertir toutes les lettres en majuscules et minuscules :

```
toupper(phrase)
```

```
## [1] "NOTES OBTENUES\nANGLAIS: 16, STAT:14, EC0=18"
tolower("AAA")
## [1] "aaa"
```

2.1.1.5 *grep()*

Permet de trouver dans un vecteur de caractères quels sont les indices des composantes du vecteur qui contiennent un ensemble de caractères. Par exemple quels sont les mots qui contiennent la lettre “e” :

```
(res <- grep(pattern = "e", x = mots))
```

```
## [1] 1 2
mots[res]
```

```
## [1] "Notes"    "obtenues"
```

On peut chercher plusieurs lettres à la fois. Ici, on cherche les mots qui contiennent une des lettres “j”, “J” et “t”. Pour cela, on utilise une expression régulière grâce aux crochets (nous verrons dans la section suivante plus en détail le fonctionnement des expressions régulières) :

```
(res <- grep(pattern = "[jSE]", x = mots))
```

```
## [1] 7 10
mots[res]
```

```
## [1] "Stat" "Eco"
```

Remarque : un vecteur de taille nulle est retourné si le critère n'est pas satisfait.

2.1.1.6 *agrep()*

Pemet de trouver dans un vecteur de caractères quels sont les indices des composantes du vecteur qui contiennent "approximativement" une sous-chaîne, l'approximation pouvant être réglée avec les options de la fonction. Dans l'exemple suivant, la lettre s minuscule est différente de S majuscule, mais cette différence d'1 caractère sera tolérée.

```
agrep("stat", mots)
```

```
## [1] 7
```

Remarque : lorsqu'on traite des fichiers de données brutes, ce fichier peut contenir des erreurs de saisie, par exemple "Toulouze" au lieu de "Toulouse" et c'est pourquoi le fait de tolérer un nombre de différences peut s'avérer intéressant.

- *sub()* pour changer une sous-chaîne de caractères par une autre :

```
(mots <- sub(pattern = "=", replacement = ":", x = phrase))
```

```
## [1] "Notes obtenues\nanglais: 16, Stat:14, Eco:18"
```

2.1.1.7 *regexp()*

Permet de dire à quelle position dans le mot se trouve une sous-chaîne de caractères. Dans l'exemple suivant, on cherche à savoir où se trouve le caractère ":" dans les mots. Si un caractère est présent, alors la fonction retourne les positions de la lettre ":" et si elle n'apparaît pas, la valeur -1 est retournée. Le résultat est encore donné sous forme de **list** car cela permet de traiter chaque mot du vecteur.

```
gregexpr(pattern = ":", text = mots, ignore.case = TRUE)
```

Compléments : en pratique, on peut vouloir faire des recherches plus complexes (plusieurs caractères, des caractères spéciaux, etc), ce qui nécessite une adaptation dans les critères de recherche. La gestion des chaînes de caractères est synthétisée dans l'aide suivante :

```
help(regexp)
```

2.1.1.8 *abbreviate()*

Pemet de faire des abbréviations de chaînes de caractères trop longue, tout en respectant l'unicité de chaque mot (autrement dit, une même abréviation ne peut pas être donnée à deux mots différents). Par exemple :

```

pays <- c("Bosnie-Herzégovine", "Burkina Faso", "Côte d'Ivoire")
abbreviate(pays)

## Warning in abbreviate(pays): abbreviate utilisé avec des caractères non ASCII
## Bosnie-Herzégovine      Burkina Faso      Côte d'Ivoire
##          "Bs-H"           "BrkF"           "Cd'I"

```

Exercice 1.1

Q1 Compter le nombre de caractères de chaque élément du vecteur suivant. Que constatez-vous ?

```
x_with_missing <- c("oui", "peut-être", NA, "non", NA, "si")
```

Q2 Dans le vecteur suivant, faire l'extraction des deux entiers qui se trouvent entre le symbole `_` et présenter le résultat sous forme de vecteur :

```

code_INSEE <- c("toulouse_31_HG", "lyon_69_Rhone",
               "marsei_13_PACA")

```

2.1.2 Les expressions régulières

Ce paragraphe s'inspire de cette note de cours écrite par Ricco Rakotomalala.

On va s'attarder sur l'utilisation d'expressions régulières qui est très populaire dans certaines disciplines et notamment le text mining qui englobe l'analyse de tweets.

2.1.2.1 Définition

Une expression régulière est une séquence de caractères qui définit un motif d'intérêt. Par exemple, on considère le motif d'intérêt “b.b.”, une séquence de 4 caractères où le 1er et 3ème caractères sont le caractère “b” et le 2ème et 4ème peuvent être n'importe quel autre caractère tel que “bobi”, “buba”, “bib1”, “bpbe”, “byb=”, etc.

On pourrait considérer un second motif d'intérêt “b.b.”, une séquence de 4 caractères où le 1er et 3ème caractère sont le caractère “b” et le 2ème et 4ème peuvent être une voyelle uniquement. Dans ce cas, “bybe” serait un candidat, mais pas “bib1”.

Une expression régulière correspond donc à la syntaxe informatique qui sera utilisée pour détecter un motif d'intérêt.

2.1.2.1.1 Exemples Par défaut, les fonctions `strsplit()`, `grep()`, `sub()` ou `regexpr()` permettent d'utiliser une expression régulière comme critère de recherche. Par exemple, pour identifier le 1er motif d'intérêt, l'expression régulière est la suivante “b.b.” où le “.” indique donc que n'importe quel caractère est accepté

```
textes <- c("bobi", "bibé", "tatane", "bAbA", "tbtc",
         "tut", "byb=", "baba", "bub1", "t5t3")
print(grep("b.b.", textes))
```

```
## [1] 1 2 4 7 8 9
```

Pour le second motif d'intérêt, on va utiliser l'expression régulière suivante :

```
print(grep("b[aeiouy]b[aeiouy]", textes))
```

```
## [1] 1 8
```

On met entre crochets les caractères qui sont autorisés.

La négation des caractères autorisés est obtenue avec le symbole “^”. Par exemple, dans l'exemple suivant, on autorise tous les caractères sauf les voyelles :

```
print(grep("b[^aeiouy]b[^aeiouy]", textes))
```

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

Pour autoriser une suite de caractères, on utilise le symbole “-”. Par exemple, si on autorise uniquement les lettres minuscules de l'alphabet, on fait :

```
print(grep("b[a-z]b[a-z]", textes))
```

```
## [1] 1 8
```

Si on autorise toutes les lettres de l'alphabet (minuscules et majuscules ainsi que les caractères spéciaux comme les accents é, à, è, ç, etc.), on utilise la syntaxe “[[:alpha:]]” entre crochets. Par exemple :

```
print(grep("b[[[:alpha:]]]b[[[:alpha:]]]", textes))
```

```
## [1] 1 2 4 8
```

Si on utilise toutes les lettres de l'alphabet ainsi que les chiffres numériques, on utilise la syntaxe “[[:alnum:]]” entre crochets. Par exemple :

```
print(grep("b[[[:alnum:]]]b[[[:alnum:]]]", textes))
```

```
## [1] 1 2 4 8 9
```

2.1.2.2 Les autres expressions régulières

Nous avons résumé différentes expressions régulières pouvant être utilisées :

- [...] : un des caractères indiqués entre les crochets. Par exemple:

```
print(grep("t[aeiouy]t[aeiouy]", textes))
```

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

- $[^...]$: tous les caractères sauf ceux indiqués après le $^$. Par exemple:

```
print(grep("t[^aeiouy]t[^aeiouy]", textes))
```

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

- $[x-y]$: les caractères compris entre x à y inclus. Par exemple :

```
print(grep("t[a-z]t[a-z]", textes))
```

```
## [1] 3 5
```

- $[:alnum:]$ équivalent à a-zA-Z0-9 avec en plus les caractères spéciaux que l'on retrouve suivant les langues utilisées comme les é, è, ù, ç, à.
- $[:alpha:]$ équivalent à a-zA-Z avec en plus les caractères spéciaux que l'on retrouve suivant les langues utilisées
- $[:digit:]$ équivalent à 0-9. Par exemple :

```
print(grep("t[[:digit:]]t[[:digit:]]", textes))
```

```
## [1] 10
```

- $[:lower:]$ équivalent à a-z avec en plus les caractères spéciaux que l'on retrouve suivant les langues utilisées
- $[:upper:]$ équivalent à A-Z avec en plus les caractères spéciaux que l'on retrouve suivant les langues utilisées comme les Â, Û, Ô, etc.
- $[:xdigit:]$ équivalent à 0-9a-fA-F
- $[:graph:]$ tout caractère graphique
- $[:print:]$ tout caractère affichable
- $[:punct:]$ tout caractère de ponctuation
- $[:blank:]$ espace, tabulation
- $[:space:]$ espace, tabulation, nouvelle ligne, retour chariot
- $[:cntrl:]$ tout caractère de contrôle

Exercice 1.2:

On considère la chaîne de caractère suivante :

```
ww <- "we went to warwick 5 times"
```

Avec la fonction *gregexpr()*, trouver les expressions régulières qui permettent de donner l'emplacement de :

- la chaîne de caractères “i”
- un chiffre numérique
- la chaîne de caractères “we”
- une des deux chaînes de caractères “w” ou “e”

- un pattern commençant par un espace suivi de la chaîne “w”.

2.1.3 Application aux tweets

On considère le vecteur suivant dont les éléments correspondent à des extraits de tweets

```
tweet <- c("TopStartupsUSA: RT @FernandoX: 7 C's of Marketing in the Digital Era.\n",
  "#Analytics #MachineLearning #DataScience #MalWare #IIoT",
  "YvesMulkers: RT @wil_bielert: RT @neptanum: Standard Model Physics from an Algebra?",
  "#BigData #Analytics #DataScience #AI #MachineLearning #IoT #IIoT #Python")
```

Exercice 1.3.

Expliquer la fonction de chacune des expressions régulières suivantes

```
correct <- gsub("(RT|via)((?:\\b\\\\W*@\\\\w+)+)", "", tweet)
correct <- gsub("@\\\\w+", "", correct)
correct <- gsub("[[:punct:]]", "", correct)
correct <- gsub("[[:digit:]]", "", correct)
correct <- gsub("http\\\\w+", "", correct)
correct <- gsub("[\\t ]{2,}", " ", correct)
correct <- gsub("^\\\\s+|\\\\s+\$", "", correct)
correct <- iconv(correct, "UTF-8", "ASCII", sub = "")
```

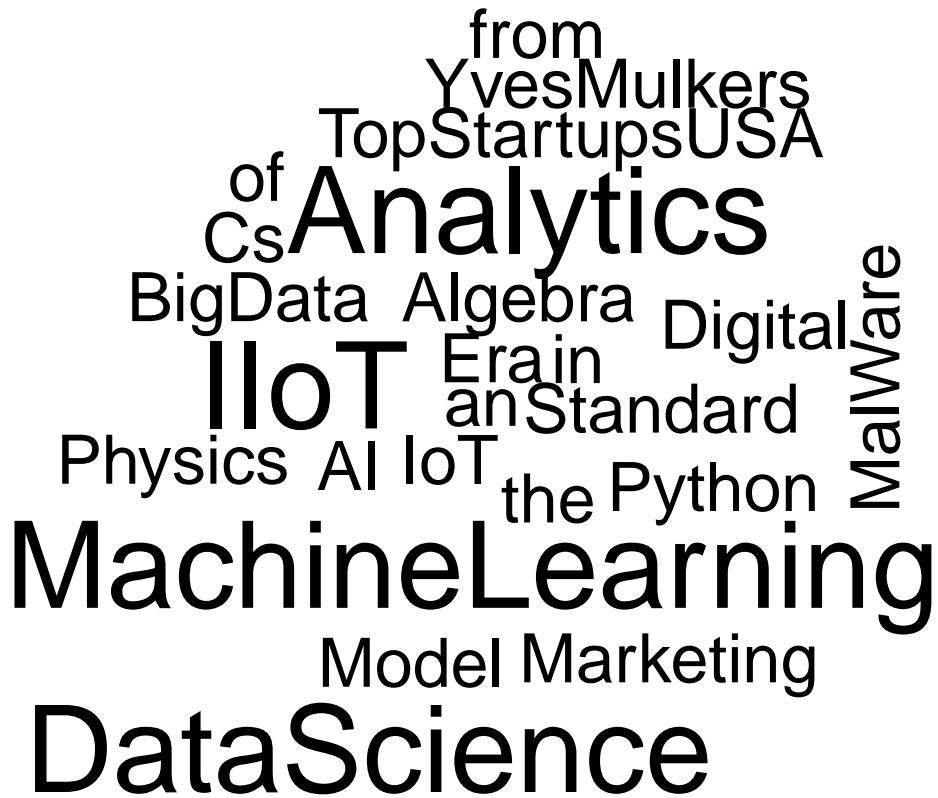
2.1.4 Nuage de mots

On présente ici la fonction `wordcloud()` du package **wordcloud** qui permet de représenter un nuage de mots en fonction du nombre d'occurrences des mots trouvés dans un corps de texte.

```
word <- unlist(strsplit(correct, " "))
tab_word <- table(word)
require("wordcloud")

## Le chargement a nécessité le package : wordcloud

## Le chargement a nécessité le package : RColorBrewer
wordcloud(names(tab_word), tab_word)
```



2.1.5 Ordonnancement

Les caractères peuvent s'ordonner comme on le fait avec des chiffres. Par exemple, le caractère “a” est plus petit que “b” qui n'est pas plus grand que “c”. Pour le vérifier :

```
"a" < "b"
## [1] TRUE
"b" > "c"
## [1] FALSE
```

On s'intéresse ici plus particulièrement aux règles d'ordonnancement utilisées pour la langue française. Selon la langue utilisée par la machine, il existe donc des règles particulières pour ordonnancer les chaînes de caractères, qui diffèrent d'une langue à l'autre. Pour vérifier la langue utilisée par la machine, on peut utiliser la commande

```
Sys.setlocale(category = "LC_CTYPE", locale = "")
## [1] "fr_FR.UTF-8"
```

Considérons la citation suivante stockée dans l'objet **citation** :

```
citation <- "Il est important que les étudiants portent un regard neuf
et irrévérencieux sur leurs études ; il ne doivent pas vénérer le savoir
mais le remettre en question (chapitre : 1 - paragraphe : 2 - ligne : 10
- page : 185. Jacob Chanowski)."
```

On peut récupérer chaque “mot” (entités séparées par des espaces”) par la commande :

```
(mots <- unlist(strsplit(citation, split = " ")))

## [1] "Il"           "est"          "important"    "que"
## [5] "les"          "étudiants"     "portent"      "un"
## [9] "regard"       "neuf"         "\net"         "irrévérencieux"
## [13] ""             "sur"          "leurs"        "études"
## [17] ";"            "il"           "ne"           "doivent"
## [21] "pas"          "vénérer"       "le"           "savoir"
## [25] "\nmais"        "le"           "remettre"     "en"
## [29] "question"     "(chapitre"    ":"           "1"
## [33] "--"           "paragraphe"   ":"           "2"
## [37] "--"           "ligne"        ":"           "10"
## [41] "\n--"          "page"         ":"           "185."
## [45] "Jacob"        "Chanowski)." "
```

Le tri des éléments (uniques) du vecteur **mots** nous montre les règles d’ordonnancement appliquées par **R**.

```
sort(unique(mots))

## [1] ""           "\n--"         "\net"         "\nmais"
## [5] "--"         ";"           ":"           "(chapitre"
## [9] "1"           "10"          "185."        "2"
## [13] "Chanowski)." "doivent"     "en"           "est"
## [17] "études"      "étudiants"    "il"           "Il"
## [21] "important"   "irrévérencieux" "Jacob"        "le"
## [25] "les"          "leurs"        "ligne"        "ne"
## [29] "neuf"         "page"         "paragraphe"   "pas"
## [33] "portent"     "que"          "question"    "regard"
## [37] "remettre"     "savoir"       "sur"          "un"
## [41] "vénérer"      ""             ""             ""
```

- les caractères spéciaux -, :, ;, (, etc. sont prioritaires sur les chiffres et les lettres.
- les chiffres sont prioritaires sur les lettres.
- les mots sont ordonnancés comme dans un dictionnaire français.
- quand il y a des lettres avec des accents, on ordonne comme s'il n'y

avait pas d'accents.

- les lettres majuscules sont insérées dans l'ordre alphabétique et ne sont pas prioritaires par rapport aux lettres minuscules.
- la syntaxe “ $|n$ ” n'est pas comptabilisée.
- les chiffres ne sont pas regardés comme des chiffres mais comme une chaîne de caractères. Autrement dit “10” doit être vu comme un mot avec 2 caractères consécutifs : “1”, puis “0”. “2” doit être vu comme 1 mot avec un seul caractère. Pour comparer ces 2 mots, on compare les caractères entre eux les uns après les autres. Dans un premier temps, on regarde le 1er caractère de chaque mot : “1” est plus petit que “2”. Aussi, quelque soit le nombre de caractère qu'on va ajouter après “1” ce mot sera plus petit que “2”. Par exemple “15552525” sera plus petit que “2”.

Exercice 1.4.

A partir du jeu de données **USArrests**, extraire les lignes dont le nom contient la chaîne de caractères “New”. Vous pouvez vous inspirer des instructions suivantes. Dans le premier cas, tous les noms de lignes contenant la lettre **C** sont renvoyés ; dans le second cas, seuls ceux commençant par **C** sont renvoyés. Consulter la fiche d'aide sur les expressions régulières pour en savoir (beaucoup !) plus (**help(regex)**).

```
USArrests[grep("C", rownames(USArrests)),]
```

```
##           Murder Assault UrbanPop Rape
## California     9.0     276      91 40.6
## Colorado       7.9     204      78 38.7
## Connecticut    3.3     110      77 11.1
## North Carolina 13.0     337      45 16.1
## South Carolina 14.4     279      48 22.5
```

```
USArrests[grep("^C", rownames(USArrests)),]
```

```
##           Murder Assault UrbanPop Rape
## California     9.0     276      91 40.6
## Colorado       7.9     204      78 38.7
## Connecticut    3.3     110      77 11.1
```

2.1.6 Package stringr

On a vu ci-dessus les fonctions de base pour manipuler des chaînes de caractères. Toutefois, si on cherche à faire des statistiques un peu plus poussées sur les chaînes de caractères, on va devoir avoir recours aux fonctions *sapply()* ou *lapply()* qui permettent de faire des opérations sur les listes. Par exemple, on cherche à calculer le nombre de fois qu'apparaît la lettre “a” dans le vecteur **mots** précédent. Pour cela, on va appliquer la fonction *sapply()* (dont nous reparlerons plus tard) sur le résultat donné par la fonction *gregexpr()*. On rappelle que le

résultat de cette fonction est -1 si le caractère n'a pas été trouvé et sinon, il retourne le vecteur des positions. Pour répondre à notre problème, on exécute donc le code suivant :

```
res1 <- gregexpr(pattern = "a", text = mots, ignore.case = T)
sapply(res1, function(x) ifelse(x[1] > 0, length(x), 0))

## [1] 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 3 0 0 0 0
## [39] 0 0 0 1 0 0 1 1
```

Le package **stringr** a vu le jour dans le but d'effacer certaines lacunes des fonctions de base et également simplifier la syntaxe des fonctions de base. Adopter les fonctions de ces packages revient un peu à oublier la syntaxe des fonctions que nous avons vues. Par exemple, la fonction *str_c()* est plus ou moins équivalente à la fonction *paste()*, la fonction *str_length()* est équivalent à la fonction *nchar()*. La fonction *str_count()* retourne le même résultat précédent en 1 seul ligne de commande :

```
library("stringr")
str_count(mots, "a")

## [1] 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 3 0 0 0 0
## [39] 0 0 0 1 0 0 1 1
```

Parmi les autres fonctions intéressantes de ce package, on notera la fonction *str_pad()* qui permet de faire en sorte qu'une chaîne de caractère (1er argument de la fonction) possède au minimum un nombre de caractère (2ème argument) et la complète si nécessaire avec un caractère (3ème argument). Par exemple, si on souhaite que chaque élément du vecteur suivant possède au moins 3 caractères et qu'on souhaite compléter les chaînes manquantes par le caractère “-” en début de chaîne, on procède ainsi :

```
vec_to_change <- c("1", "10", "105", "9999", "0008")
str_pad(vec_to_change, 4, pad = "0")

## [1] "0001" "0010" "0105" "9999" "0008"
```

On notera que le package **stringr** a été développé par Hadley Wickam (**RStudio**) qui est l'auteur d'un grand nombre d'autres packages avec cet esprit de rendre les choses plus simples pour l'utilisateur. Nous vous présenterons ces outils au fur et à mesure, mais à notre sens, il est important d'avoir conscience que derrière ces fonctions, se cachent des programmes qui utilisent les fonctions de base de **R**.

On citera également le package **stringi** qui propose un grand nombre de fonctions plus orientées vers l'analyse statistique de chaînes de caractères.

Bibliographie On pourra consulter ce document très intéressant et complet sur la gestion des chaînes de caractères avec **R** (la version gratuite est très bien) : <https://leanpub.com/r4strings>

2.1.7 Package glue

Ce package contient la fonction `glue()` qui permet d'insérer dans du texte des objets qui ont été créés dans l'environnement courant. En reprenant l'exemple de l'auteur du package (<https://cran.r-project.org/web/packages/glue/readme/README.html>) :

```
require("glue")

## Le chargement a nécessité le package : glue
name <- "Fred"
anniversary <- as.Date("1991-10-12")
age <- as.numeric(floor((Sys.Date() - anniversary)/365))
new_object <- glue('My name is {name},',
  ' my age next year is {age + 1},',
  ' my anniversary is {format(anniversary, "%A, %d %B, %Y")}.')
```

Remarque: l'objet créé est à la fois un objet de type **glue** et **character** en même temps. La particularité de ce type d'objets est qu'il hérite de toutes les fonctions qui peuvent s'appliquer à ces deux types:

```
class(new_object)

## [1] "glue"      "character"
new_object

## My name is Fred, my age next year is 31, my anniversary is samedi, 12 octobre, 1991

Il est important de noter qu'une fois l'objet glue créé, sa valeur est fixée. Autrement dit, même si on modifie les objets qui ont été utilisés pour le créer, cela ne le modifiera (à moins bien sûr de re-exécuter la commande). Exemple :

name <- "Jojo"
new_object

## My name is Fred, my age next year is 31, my anniversary is samedi, 12 octobre, 1991

new_object <- glue('My name is {name},',
  ' my age next year is {age + 1},',
  ' my anniversary is {format(anniversary, "%A, %d %B, %Y")}.')
new_object

## My name is Jojo, my age next year is 31, my anniversary is samedi, 12 octobre, 1991
```

2.2 Les facteurs

Même s'ils peuvent a priori ressembler à des chaînes de caractères, les **factor** ont un comportement différent. Cette classe d'objet a été créé pour correspondre à une variable qualitative.

On commence par créer un vecteur de chaîne de caractères :

```
genre <- sample(c("Ctrl", "Trait"), size = 10000, replace = TRUE)
```

Puis, on le transforme en **factor** en utilisant la fonction *factor()* ou *as.factor()* :

```
genre_fact <- factor(genre)
```

Les facteurs ont des modalités pré-définies qui sont retournées avec la fonction *levels()*. L'affectation d'une valeur différente de ces modalités pré-définies provoque un message d'avertissement et une valeur manquante dans le vecteur :

```
levels(genre_fact)
```

```
## [1] "Ctrl"  "Trait"  
genre_fact[1] <- "Autre"
```

```
## Warning in `<-.factor`(`*tmp*`, 1, value = "Autre"): niveau de facteur  
## incorrect, NAs générés
```

```
genre_fact[1]
```

```
## [1] <NA>  
## Levels: Ctrl Trait
```

ce qui n'est bien entendu pas le cas pour les vecteurs de caractères :

```
genre[1] <- "Autre"  
genre[1]
```

```
## [1] "Autre"
```

Dans le cas de vecteurs de taille importantes, le stockage d'un **factor** est un peu moins volumineux qu'un objet **character**. Ici, on utilise la fonction *object.size()* qui indique la taille allouée à un objet en mémoire vive :

```
object.size(genre)
```

```
## 80216 bytes  
object.size(genre_fact)
```

```
## 40560 bytes
```

L'exemple suivant permet de mieux comprendre qu'un **factor** peut être considéré comme un vecteur de valeurs entières où chaque entier pourrait être remplacé par un **label**. Dans le cas où l'on souhaite associer un **label**, on procède de la façon suivante.

```

vec <- sample(1:4, size = 20, rep = T)
(f_vec <- factor(vec, levels = 1:3,
                  labels = c("Rien", "Peu", "Beaucoup")))

## [1] Peu      Peu      <NA>     <NA>     Peu      Rien     Peu      Rien
## [9] Beaucoup <NA>     <NA>     Peu      Peu      Rien     Peu      Rien
## [17] Beaucoup Beaucoup <NA>     Rien
## Levels: Rien Peu Beaucoup

```

Remarque : si on oublie d'associer un **level** à un **label**, cela a pour conséquence de créer une valeur manquante.

La fonction *cut()* qui permet le recodage d'une variable quantitative en classes, génère un objet de type **factor**. On précise les amplitudes des classes avec l'option **breaks** :

```

set.seed(123)
mesures <- rnorm(100)
codage <- cut(mesures, breaks = -4:4)
table(codage)

## codage
## (-4,-3] (-3,-2] (-2,-1] (-1,0] (0,1] (1,2] (2,3] (3,4]
##       0       1      13     34     35     14      3      0

```

Pour aider à trouver un découpage d'une variable quantitative, la fonction *classIntervals()* du package **classInt** propose différentes méthodes de discrétilisation d'une variable quantitative. Parmi ces méthodes, on compte la méthode basée sur des classes d'amplitudes égales, d'effectifs égaux, ou calculées à partir de l'algorithme des *k-means* :

```

require("classInt")
codage2 <- cut(mesures,
               classIntervals(mesures, n = 5, style = "kmeans")$brks)
table(codage2)

## codage2
##  (-2.31,-0.874] (-0.874,-0.0726]  (-0.0726,0.614]      (0.614,1.44]
##           13                 31                 28                 19
##  (1.44,2.19]
##             8

```

Remarque : les fonctions classiques d'importation des jeux de données codent les variables qualitatives sous forme de **factor**. Cet aspect a été critiqué par certains programmeurs dont Hadley Wickam (nous en verrons les raisons un peu plus tard), qui préconise un codage des variables qualitatives sous forme de **character**. Toutefois, un des avantages de la classe **factor** est que cela permet de représenter les modalités d'une variable qualitative de façon ordonnée, ce qui se révèlera très pratique pour représenter des diagramme en barres par exemple..

Exercice 1.5.

- **Q1. Facteur ordonné:**
 - Créer un vecteur de chaîne de caractères **doses** de taille 25, comprenant les valeurs “faible”, “moyenne” ou “forte” (on pourra créer ce vecteur de façon aléatoire).
 - Convertir cet élément en **f_dose** de type **factor** ordonné (voir si besoin *?factor*).
 - Vérifier que les niveaux du facteur sont effectivement ordonnés. Pour cela, il suffit de comparer d’utiliser les opérateurs de comparaison < ou > sur deux composantes du vecteur.
- **Q2. Codage et comptage :** donner un équivalent de l’enchaînement des fonctions *cut()* et *table()* utilisées précédemment.

2.3 Les dates

2.3.1 Dates et unités de temps

Il existe plusieurs packages pour manipuler des données temporelles ; voir la Task View Time Series Analysis. Nous nous contenterons ici de présenter quelques manipulations élémentaires. Une référence sur le sujet :

- G. Grothendieck and T. Petzoldt (2004), **R** Help Desk: Date and Time Classes in **R**, **R** News 4(1), 29-32 (https://www.r-project.org/doc/Rnews/Rnews_2004-1.pdf).

Dans **R**, une façon naturelle de manipuler des dates est d’utiliser la classe d’objet **Date**. Voici l’allure d’un objet de classe **Date** :

```
(format.Date <- Sys.Date())
```

```
## [1] "2021-10-15"
```

```
class(format.Date)
```

```
## [1] "Date"
```

En gros, il s’agit d’une chaîne de caractère de la forme “**YYYY-MM-DD**”, parfois “**YYYY/MM/DD**” ou encore “**MM/DD/YY**”. Pour passer d’une chaîne de caractère à un objet de classe **Date**, il faut donc préciser où sont placés les années, mois et jours dans la chaîne de caractère, préciser si les mois sont des valeurs numériques ou écrit en lettre, etc. Par exemple :

```
dates <- c("01/01/17", "02/03/17", "03/05/17")
as.Date(dates, "%d/%m/%y")
```

```
## [1] "2017-01-01" "2017-03-02" "2017-05-03"
```

```
dates <- c("1 janvier 2017", "2 mars 2017", "3 mai 2017")
as.Date(dates, "%d %B %Y")
```

```
## [1] "2017-01-01" "2017-03-02" "2017-05-03"
```

Les informations sur la façon dont convertir les chaînes de caractères en objet **Date** sont données dans l'aide suivante :

```
?format.Date
```

Une autre classe d'objet utile est la classe **POSIXct/POSIXlt**. Le format **POSIXct/POSIXlt** est plus précis que le format **Date** car il mesure à la fois la date et l'heure. Ce format est notamment utilisé dans les séries temporelles d'indices boursiers.

```
(format.POSIXlt <- Sys.time())
```

```
## [1] "2021-10-15 20:11:03 CEST"
```

```
class(format.POSIXlt)
```

```
## [1] "POSIXct" "POSIXt"
```

Un certain nombre de fonctions de **R** reconnaissent ce type de format. On en cite ici quelques-unes :

```
weekdays(format.POSIXlt)
```

```
## [1] "vendredi"
```

```
months(format.POSIXlt)
```

```
## [1] "octobre"
```

```
quarters(format.POSIXlt)
```

```
## [1] "Q4"
```

De même que pour les dates, il est possible de convertir des chaînes de caractères en **POSIXct/POSIXlt** ou de faire l'inverse. Pour cela, il faut respecter la nomenclature des dates (voir l'aide en ligne `?strptime`). Pour convertir une chaîne de caractères en **POSIXct/POSIXlt** :

```
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
```

```
times <- c("23:03:20", "22:29:56", "01:03:30", "18:21:03", "16:56:26")
```

```
x <- paste(dates, times)
```

```
strptime(x, "%m/%d/%y %H:%M:%S")
```

```
## [1] "1992-02-27 23:03:20 CET" "1992-02-27 22:29:56 CET"
```

```
## [3] "1992-01-14 01:03:30 CET" "1992-02-28 18:21:03 CET"
```

```
## [5] "1992-02-01 16:56:26 CET"
```

Pour faire l'inverse (transformer un **POSIXct/POSIXlt** en **character**) :

```
(z <- Sys.time())
```

```
## [1] "2021-10-15 20:11:03 CEST"
```

```
format(z, "%a %d %b %Y %X %Z")
## [1] "ven. 15 oct. 2021 20:11:03 CEST"
```

La fonction `system.time()` renvoie plusieurs informations concernant le temps de calcul d'une commande :

- *Utilisateur* : il s'agit du temps mis par l'ordinateur pour exécuter directement le code donné par l'utilisateur,
- *Système* : il s'agit du temps utilisé pas directement par le calcul, mais par le système (ex: gestion des entrées/sorties, écriture sur le disque, etc.) lié au code qui doit être exécuté.
- *écoulé* : il s'agit du temps Utilisateur + Système C'est en général ce dernier qui est utilisé pour comparer des temps de calcul.

```
system.time(for(i in 1:100) var(runif(100000)))
## utilisateur     système     écoulé
##      0.357        0.024      0.381
```

Remarque : il est parfois compliqué de convertir des chaînes de caractères en **Date** ou **POSIXct/POSIXlt**, mais une fois que cela est fait, il est alors possible d'utiliser un nombre conséquent de packages existants, qui permettent en autre la manipulation de séries temporelles.

Exercice 1.6.

Q1 Quel jour (lundi, mardi ... ?) sera le 1er janvier de l'année 2022 ?

Q2 Combien de jours nous séparent du 31 décembre de l'année en cours

2.3.2 Séries temporelles

Nous faisons ici un aparte pour évoquer la manipulation de séries temporelles.

On commence par simuler deux séries temporelles issues respectivement d'un processus AR(2) et MA(2). Nous ne rentrerons pas dans le détail de ces modèles mais le lecteur pourra se référer à l'ouvrage d'Yves Aragon "Séries temporelles avec R" pour une introduction. On utilise la fonction `set.seed()` avant chaque simulation, ce qui va nous permettre de générer la même séquence dès lors qu'on utilisera les mêmes entiers comme argument de la fonction (493 et 494 ici).

```
set.seed(493)
x1 <- arima.sim(model = list(ar = c(.9, -.2)), n = 100)
set.seed(494)
x2 <- arima.sim(model = list(ma = c(-.7, .1)), n = 100)
```

Le principe d'une série temporelle est que les observations sont associées à une date et dans certains cas une date et un temps (indices boursiers observés en

temps réel). Pour faire simple, on va associer la série à des dates journalières commençant le 1er octobre 2017 et de taille 100. Une façon de faire est d'utiliser la fonction `seq.Date()` :

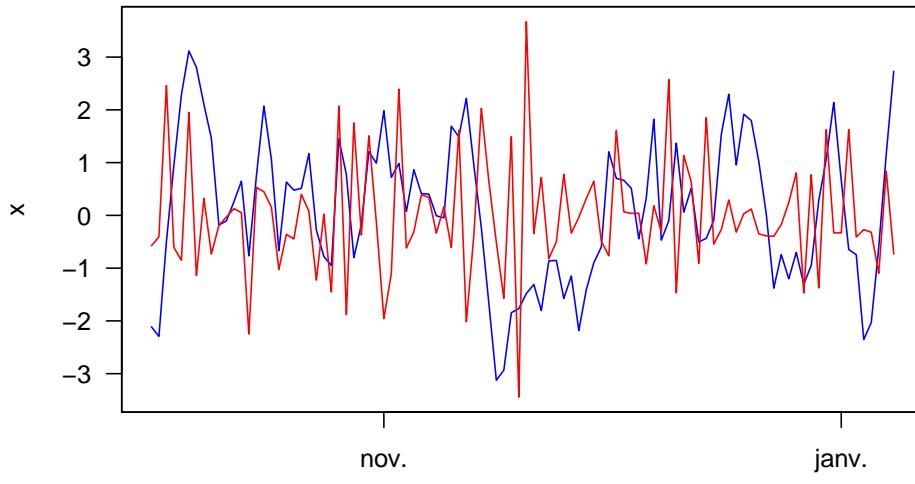
```
date_x <- seq.Date(as.Date("2017-10-01"), by = "day", len = 100)
```

Ensuite, on utilise la fonction `zoo()` du package qui porte le même nom pour associer les séries aux dates précédemment créées :

```
require("zoo")
x <- zoo(cbind(x1, x2), date_x)
```

L'objet précédemment créé peut ensuite être appliquée aux fonctions du package **zoo**. Ainsi, en appliquant la fonction `plot()` à un tel objet, cela a pour avantage d'afficher en abscisses les dates, d'afficher plusieurs courbes s'il s'agit de séries temporelles multidimensionnelles, etc. On pourra consulter les fonctions du package **zoo** pour plus d'informations (`help(package="zoo")`)

```
par(las = 1)
plot(x, col = c("blue", "red"), screens = 1)
```



Remarque : l'option `las = 1` dans la fonction `par()` permet d'afficher la légende de l'axe des ordonnées horizontalement. L'option `screens = 1` permet de représenter les deux séries dans la même figure.

2.4 Opérations ensemblistes

On définit deux vecteurs A et B d'entiers.

```
(A <- 1:10)
```

```
## [1] 1 2 3 4 5 6 7 8 9 10
(B <- c(3:6, 12, 15, 18))
```

```
## [1] 3 4 5 6 12 15 18
```

Les opérations ensemblistes classiques sont :

2.4.1 L’union

```
union(A, B)
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 12 15 18
```

équivalent à la syntaxe suivante écrite avec des fonctions de base:

```
unique(c(A, B))
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 12 15 18
```

2.4.2 L’intersection

- fonction *intersect()*

```
intersect(A, B)
```

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

équivalent à la syntaxe suivante écrite avec des fonctions de base:

```
A[A %in% B]
```

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

- fonction *match()*

Ici, il nous semble important de parler de la fonction *match()* qui a été utilisée pour coder la fonction *intersect()*. Dans le code ci-dessous, elle retourne pour chaque élément de **A** s'il se trouve dans **B** et si oui à quelle position dans **B**. Ci-dessous, le 3ème élément de **A** est bien dans **B** et il se situe à la 1ère position de **B**.

```
match(A, B)
```

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

Il est important de rappeler que l'opérateur `%in%` fait appel à la fonction *match()*. Pour afficher le code de la fonction `%in%`, on peut utiliser la syntaxe suivante:

```
`%in%`
```

```
## function (x, table)
```

```
## match(x, table, nomatch = 0L) > 0L
## <bytecode: 0x55cc3b5e2548>
## <environment: namespace:base>
```

2.4.3 La différence ($A - B$, différent de $B - A$)

```
setdiff(A, B)

## [1] 1 2 7 8 9 10
setdiff(B, A)

## [1] 12 15 18
```

Pour savoir si un ou plusieurs éléments sont contenus dans un ensemble :

```
is.element(2, A)

## [1] TRUE
is.element(2, B)

## [1] FALSE
is.element(A, B)

## [1] FALSE FALSE TRUE TRUE TRUE TRUE FALSE FALSE FALSE
is.element(B, A)

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

Attention : on utilise ces fonctions sur des objets de même classe. Si on fait l'union d'un vecteur d'entiers avec un vecteur de caractères, le vecteur d'entiers sera transformé en caractères. Par exemple :

```
let <- letters[1:10]
union(A, let)

## [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
## [16] "f"  "g"  "h"  "i"  "j"
```

Il est donc possible de faire des opérations ensemblistes sur les chaînes de caractères.

Exercice 1.7.

Q1 : donner une notation équivalente à `is.element(2, A)`.

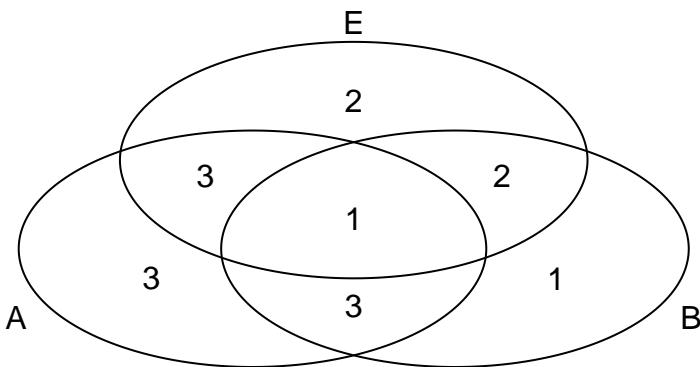
Q2 : l'objet `letters` est un vecteur de longueur 26 qui contient les lettres de l'alphabet. Tester l'appartenance des lettres `k` et `m` à l'alphabet ; que renvoie la syntaxe “réciproque” (appartenance de l'alphabet à l'ensemble `c("k", "m")`) ? Comment obtenir le rang des lettres `k` et `m` dans l'alphabet ?

Q3 : en plus des 2 vecteurs **A** et **B** définis précédemment, considérons le vecteur **E** (nous évitons la lettre **C** qui existe déjà dans **R**, voir *help(C)*) suivant :

```
E <- c(18, 1, 9, 14, 12, 6, 2, 19)
```

Donner l'enchaînement des commandes qui permettent de retrouver les valeurs disposées sur le diagramme de Venn ci-dessous (obtenu avec la fonction *venn()* du package **gplots**).

```
require("gplots")
venn(list(A = A, B = B, E = E))
```



2.5 Manipulation de bases de données

2.5.1 Jointure et agrégation

On considère une table **patient** qui contient des informations sur les patients et une table **visite** qui contient des informations sur les visites. La clé de référence est le nom du patient dont la variable ne porte pas le même nom dans les deux tables.

```
patient <- data.frame(
  nom.famille = c("René", "Jean", "Ginette", "Joseph"),
  ddn = c("02/02/1925", "03/03/1952", "01/10/1992", "02/02/1920"),
  sexe = c("m", "m", "f", "m"))

visite <- data.frame(
  nom = c("René", "Jean", "René", "Simone", "Ginette"),
  ddv = c("01/01/2020", "10/12/2020", "05/01/2020", "04/12/2020", "05/10/2020"))
```

Problématique : on souhaite avoir une table contenant l'âge du patient au moment de sa visite. Pour cela, on voit bien qu'il faut connaître la date de naissance du patient au moment de sa visite. L'idée est donc d'ajouter une colonne “ddn” à la table **visite** pour pouvoir ensuite calculer l'âge en faisant la différence entre la date de visite et la date naissance.

2.5.1.1 Jointure sans utiliser la fonction *merge()*

Dans un premier temps, nous allons essayer de répondre au problème sans utiliser la fonction *merge()*. Pour cela, nous allons utiliser la fonction *match()* vue précédemment. Elle va nous permettre de savoir où se trouvent dans la table **patient**, les patients qui ont eu des visites. Une fois les indices connus, on a plus qu'à sélectionner les dates de naissance des patients.

```
visite$ddn <- patient$ddn[match(visite$nom, patient$nom.famille)]
```

Ensuite, on calcule l'âge du patient :

```
visite$age <- round(as.numeric(as.Date(visite$ddv, format = "%d/%m/%Y") -  
                      as.Date(visite$ddn, format = "%d/%m/%Y"))/365,  
                     0)
```

Enfin, on efface la date de naissance qui ne nous intéresse pas ici :

```
visite$ddn <- NULL  
head(visite)
```

```
##      nom      ddv age  
## 1 René 01/01/2020  95  
## 2 Jean 10/12/2020  69  
## 3 René 05/01/2020  95  
## 4 Simone 04/12/2020 NA  
## 5 Ginette 05/10/2020  28
```

Pour la suite, on remet à jour la table **visite** :

```
visite$age <- NULL
```

2.5.1.2 Jointure en utilisant la fonction *merge()*

A présent, nous allons utiliser la fonction *merge()* qui permet de réaliser la jointure entre deux tables. Il est essentiel de préciser la clé de référence des deux tables avec l'option **by=** si les deux tables ont un nom de clé identique ou alors **by.x=** et **by.y=** si la clé de référence porte un nom différent selon la table.

Dans la première commande, le merge se fait sur les variables qui sont présentes dans les deux tables. Autrement dit, l'individu "Simone" n'est pas présente dans la nouvelle table compte tenu qu'elle n'est pas identifiée parmi les patients.

```
visite$age <- NULL  
merge(visite, patient, by.x = "nom", by.y = "nom.famille")
```

```
##      nom      ddv      ddn sexe  
## 1 Ginette 05/10/2020 01/10/1992   f  
## 2 Jean 10/12/2020 03/03/1952   m  
## 3 René 01/01/2020 02/02/1925   m
```

```
## 4 René 05/01/2020 02/02/1925 m
```

Si on souhaite garder toute l'information contenue dans le fichier visite, on ajoute l'option **all.x=TRUE**. Autrement dit, on affiche ici toutes les visites, y compris celles des personnes qui ne sont pas dans la table **patient**. On remarquera que Simone apparaît en queue de table :

```
merge(visite, patient, by.x = "nom", by.y = "nom.famille", all.x = TRUE)

##      nom      ddv      ddn sexe
## 1 Ginette 05/10/2020 01/10/1992 f
## 2 Jean   10/12/2020 03/03/1952 m
## 3 René   01/01/2020 02/02/1925 m
## 4 René   05/01/2020 02/02/1925 m
## 5 Simone 04/12/2020 <NA> <NA>
```

Enfin, si on souhaite conserver l'information dans les deux tables, on utilise **all.x=TRUE** et **all.y=TRUE** :

```
(visite.patient <- merge(visite, patient, by.x = "nom",
                         by.y = "nom.famille", all.x = TRUE, all.y = TRUE))

##      nom      ddv      ddn sexe
## 1 Ginette 05/10/2020 01/10/1992 f
## 2 Jean   10/12/2020 03/03/1952 m
## 3 Joseph  <NA> 02/02/1920 m
## 4 René   01/01/2020 02/02/1925 m
## 5 René   05/01/2020 02/02/1925 m
## 6 Simone 04/12/2020 <NA> <NA>
```

Pour calculer l'âge du patient au moment de la visite, on utilise la commande vue précédemment :

```
visite.patient$age <- round(as.numeric(as.Date(visite.patient$ddv,
                                                format = "%d/%m/%Y")) -
                           as.Date(visite.patient$ddn,
                                                format = "%d/%m/%Y"))/365,
                           0)
```

Maintenant, on souhaite connaître l'âge des patients lors de leurs visites en fonction du sexe. On va donc faire une aggrégation.

2.5.1.3 Aggrégation

Pour faire l'agrégation, on utilise la fonction *tapply()* pour aggréger une variable ou *aggregate()* pour plusieurs variables. Le principe de ces fonctions est le suivant :

- 1. utiliser *split()* sur l'échantillon total pour découper en sous-échantillons selon la variable utilisée pour faire l'aggrégation:

```
my_split <- split(x = visite.patient$age, f = visite.patient$sex)
```

- 2. utiliser *lapply()* ou *sapply()* pour calculer des statistiques sur chaque sous-échantillon

```
sapply(my_split, FUN = mean, na.rm = T)
```

```
##           f          m
## 28.00000 86.33333
```

En utilisant la fonction *tapply()*, les deux étapes précédentes sont réalisées à la suite :

```
tapply(X = visite.patient$age, INDEX = list(sexe = visite.patient$sex),  
       FUN = mean, na.rm = T)
```

```
## sexe
##       f          m
## 28.00000 86.33333
```

Si on utilise la fonction *aggregate()*, la variable qui permet d'aggrégner doit être mise sous forme d'une liste dans l'option **by**=. L'option **FUN**= renseigne s'il s'agit de faire une somme, une moyenne, etc. Dans notre cas :

```
aggregate(visite.patient$age, by = list(S = visite.patient$sex),  
          FUN = mean, na.rm = TRUE)
```

```
##   S      x
## 1 f 28.00000
## 2 m 86.33333
```

Exercice 1.8.

Q1 à partir du jeu de données **iris** (disponible par défaut dans l'environnement courant), construire l'objet **iris1** qui contient en fonction des espèces (variable **Species**), la taille moyenne de la variable **Petal.Length**.

Q2 Construire l'objet **iris2** qui contient en fonction des espèces, la taille totale de la variable **Petal.Width**.

Q3 Faire le merge des jeux de données **iris1** et **iris2**.

2.5.2 Package dplyr

Ce document est inspiré d'une présentation de Sophie Lamarre aux rencontres des Ingénieurs statisticiens de Toulouse, disponible sur ce lien. On recommande aussi la lecture de la vignette du package, disponible sur le site du CRAN : <https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html>

Ce package est de plus en plus utilisé dans la manipulation de jeu de données, notamment parmi les "Data Scientists". Son principe est de simplifier la syntaxe

des fonctions de base de **R** et d'utiliser du code **C++** derrière certaines de ses fonctions dans le but d'améliorer sensiblement les temps de calcul. Il fait partie du projet Tidyverse qui contient un ensemble de packages, développés en grande partie par **RStudio**. Avec la commande suivante, vous chargez un ensemble de packages dont certains très populaires (**ggplot2**, **dplyr**, etc.)

```
require("tidyverse")

## Le chargement a nécessité le package : tidyverse

## -- Attaching packages ----- tidyverse 1.3.1 --

## v ggplot2 3.3.3      v purrr    0.3.4
## v tibble   3.1.2      v dplyr    1.0.6
## v tidyverse 1.1.3      vforcats  0.5.1
## v readr    1.4.0

## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::collapse() masks glue::collapse()
## x dplyr::filter()   masks stats::filter()
## x dplyr::lag()      masks stats::lag()
```

Description du jeu de données utilisées : il s'agit du jeu de données **diamonds** du package **ggplot2**. On observe sur un peu plus de 50000 diamants, un certain nombre de variables dont : le prix de vente, le poids, la qualité, la couleur, etc.

head(diamonds)

```
## # A tibble: 6 x 10
##   carat cut      color clarity depth table price     x     y     z
##   <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal     E     SI2     61.5     55     326  3.95  3.98  2.43
## 2  0.21 Premium   E     SI1     59.8     61     326  3.89  3.84  2.31
## 3  0.23 Good      E     VS1     56.9     65     327  4.05  4.07  2.31
## 4  0.29 Premium   I     VS2     62.4     58     334  4.2   4.23  2.63
## 5  0.31 Good      J     SI2     63.3     58     335  4.34  4.35  2.75
## 6  0.24 Very Good J     VVS2    62.8     57     336  3.94  3.96  2.48
```

Voici le résumé statistique des variables :

```
summary(diamonds)
```

```

##      carat          cut      color     clarity       depth
## Min.  :0.2000    Fair      : 1610   D: 6775   SI1     :13065   Min.  :43.00
## 1st Qu.:0.4000   Good     : 4906   E: 9797   VS2     :12258   1st Qu.:61.00
## Median :0.7000   Very Good:12082  F: 9542   SI2     : 9194   Median :61.80
## Mean   :0.7979   Premium  :13791   G:11292   VS1     : 8171   Mean   :61.75
## 3rd Qu.:1.0400   Ideal    :21551   H: 8304   VVS2    : 5066   3rd Qu.:62.50
## Max.   :5.0100
##
```

```

##      table      price          x          y
## Min.   :43.00   Min.   : 326   Min.   : 0.000   Min.   : 0.000
## 1st Qu.:56.00   1st Qu.: 950   1st Qu.: 4.710   1st Qu.: 4.720
## Median :57.00   Median :2401    Median : 5.700   Median : 5.710
## Mean   :57.46   Mean   :3933    Mean   : 5.731   Mean   : 5.735
## 3rd Qu.:59.00   3rd Qu.:5324    3rd Qu.: 6.540   3rd Qu.: 6.540
## Max.   :95.00   Max.   :18823   Max.   :10.740   Max.   :58.900
##
##      z
## Min.   : 0.000
## 1st Qu.: 2.910
## Median : 3.530
## Mean   : 3.539
## 3rd Qu.: 4.040
## Max.   :31.800
##

```

Remarque : le jeu de données **diamonds** est dans un format de données nouveau : **tibble**. Il s'agit encore une fois d'une nouveauté proposée par **RStudio**. L'objet de classe **data.frame** peut présenter certaines contraintes pour ses utilisateurs. Par exemple, lorsque'on souhaite afficher un **data.frame** dans la console, **R** affiche autant de lignes qu'il le peut. D'autre part, avec un **data.frame**, certains noms de variables ne sont pas autorisés. Par exemple, dans un **data.frame** un nom de variable ne peut pas commencer par un chiffre ce qui n'est pas le cas avec le **tibble** :

```

data.frame(`1a` = 1:10)
tibble(`1a` = 1:10)

```

C'est pourquoi des développeurs ont pensé à créer un nouveau type d'objets, le **tibble** qui ne présenterait pas ce genre de contraintes. Ce n'est pas une grande révolution, mais ce package faisant partie du projet **tidyverse**, c'est essentiellement ce type de données qui est utilisé dans cet univers. Les fonctions que nous allons présenter dans cette section s'appliquent aussi bien sur des **data.frame** que sur des **tibble**, ce dernier type d'objet ayant hérité des propriétés des **data.frame**. Pour en savoir plus sur les **tibbles**, vous pouvez consulter ce cours en ligne : <http://r4ds.had.co.nz/tibbles.html>.

2.5.2.1 Sélection de lignes avec la fonction *filter()*

La fonction “phare” du package **dplyr** est la fonction *filter()* qui permet de sélectionner un sous-échantillon du jeu de données à partir de critères qu'on lui donne. Par exemple, pour sélectionner uniquement les diamants d'une valeur supérieure à 15000 dollars et ayant une couleur **E** ou **F**, on écrit :

```
filtrage1 <- dplyr::filter(diamonds, price > 15000 & (color == "E" | color == "F"))
```

Ou de façon équivalente (les virgules remplaçant la condition **et**) :

```
filtrage1 <- filter(diamonds, price > 15000, (color == "E" | color == "F"))
head(filtrage1, 3)

## # A tibble: 3 x 10
##   carat cut      color clarity depth table price     x     y     z
##   <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  1.54 Premium E      VS2       62.3   58 15002  7.31  7.39  4.58
## 2  1.19 Ideal   F      VVS1      61.5   55 15005  6.82  6.84  4.2
## 3  2.05 Very Good F      SI2       61.9   56 15017  8.13  8.18  5.05
```

Une nouveauté est également l'utilisation de l'opérateur `%>%` dit 'pipe' en anglais. Il se trouve après un **data.frame** et indique qu'on va utiliser une fonction dont le premier argument est un objet de type **data.frame**. Dans ce cas, ce n'est plus la peine d'indiquer le premier argument de la fonction `filter()`. Nous verrons par la suite son intérêt lorsqu'on souhaite appliquer successivement des commandes.

```
filtrage1 <- diamonds %>%
  filter(price > 15000, (color == "E" | color == "F"))
head(filtrage1, 3)

## # A tibble: 3 x 10
##   carat cut      color clarity depth table price     x     y     z
##   <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  1.54 Premium E      VS2       62.3   58 15002  7.31  7.39  4.58
## 2  1.19 Ideal   F      VVS1      61.5   55 15005  6.82  6.84  4.2
## 3  2.05 Very Good F      SI2       61.9   56 15017  8.13  8.18  5.05
```

Remarque: sans utiliser le package **dplyr**, on aurait du faire quelque chose comme ça :

```
filtrage1 <- with(diamonds, diamonds[price > 15000 &
                                         (color == "E" | color == "F"), ])
```

On aurait également pu utiliser la fonction de base `subset()` dont le package **dplyr** s'est fortement inspiré.

```
filtrage1 <- subset(diamonds, price > 15000 & (color == "E" | color == "F"))
```

2.5.2.2 Trier le jeu de données avec la fonction `arrange()`

On souhaite trier le jeu de données en fonction de la coupe, de la couleur et du prix (en valeur décroissante). Pour cela, on fait :

```
tri1 <- arrange(diamonds, cut, color, desc(price))
head(tri1, 3)
```

```
## # A tibble: 3 x 10
##   carat cut      color clarity depth table price     x     y     z
##   <dbl> <ord>    <ord> <ord>   <dbl> <dbl> <dbl> <dbl> <dbl>
```

```
##   <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  2.02 Fair D      SI1       65     55 16386  7.94  7.84  5.13
## 2  2.01 Fair D      SI2       66.9    57 16086  7.87  7.76  5.23
## 3  3.4  Fair D      I1        66.8    52 15964  9.42  9.34  6.27
```

Sans le package **dplyr**, on aurait fait :

```
tri1 <- with(diamonds, diamonds[order(cut, color, -price),])
```

2.5.2.3 Sélectionner certaines variables avec la fonction *select()*

On ne souhaite garder que les variables **carat**, **price**, **color**, **z** et les variables dont le label contient le caractères **i** :

```
selection1 <- select(diamonds, carat, price, color, z,
                      dplyr::contains("i"))
head(selection1, 3)
```

```
## # A tibble: 3 x 5
##   carat price color     z clarity
##   <dbl> <int> <ord> <dbl> <ord>
## 1 0.23   326 E      2.43 SI2
## 2 0.21   326 E      2.31 SI1
## 3 0.23   327 E      2.31 VS1
```

Sans le package **dplyr**, on aurait fait :

```
selection1 <- diamonds[, unique(c("carat", "price", "color", "z",
                                    names(diamonds)[grep("i", names(diamonds))]))]
```

Remarque : dans la deuxième syntaxe, pour que la variable **price** n'apparaisse qu'une seule fois dans le jeu de données, on a du utiliser la fonction *unique()*.

2.5.2.4 Changer le nom de variables avec la fonction *rename()*

On souhaite changer le nom de la variable **z** par le label **width** et **color** par **code_color** :

```
renom1 <- rename(diamonds, width = z, code_color = color)
head(renom1, 3)
```

```
## # A tibble: 3 x 10
##   carat cut     code_color clarity depth table price     x     y width
##   <dbl> <ord>   <ord>      <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1 0.23 Ideal    E          SI2      61.5    55  326  3.95  3.98  2.43
## 2 0.21 Premium  E          SI1      59.8    61  326  3.89  3.84  2.31
## 3 0.23 Good     E          VS1      56.9    65  327  4.05  4.07  2.31
```

Sans le package **dplyr**, on aurait fait :

```
names(diamonds)[c(match("z", names(diamonds)),
                  match("color", names(diamonds)))] <-
  c("width", "code_color")
```

2.5.2.5 Ajouter une nouvelle colonne avec la fonction *mutate()*

On calcule le prix au kilo pour chaque diamant et on convertit en euros :

```
calcul1 <- mutate(diamonds, prix.kilo = price / carat,
                   prix.kilo.euro = prix.kilo * 0.9035)
head(calcul1, 3)
```

```
## # A tibble: 3 x 12
##   carat cut      code_color clarity depth table price     x     y width prix.kilo
##   <dbl> <ord>    <ord>      <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl> <dbl>
## 1  0.23 Ideal     E          SI2      61.5    55   326  3.95  3.98  2.43   1417.
## 2  0.21 Premium   E          SI1      59.8    61   326  3.89  3.84  2.31   1552.
## 3  0.23 Good      E          VS1      56.9    65   327  4.05  4.07  2.31   1422.
## # ... with 1 more variable: prix.kilo.euro <dbl>
```

Sans le package **dplyr**, on aurait fait :

```
diamonds$prix.kilo <- diamonds$price/diamonds$carat
diamonds$prix.kilo.euro <- diamonds$prix.kilo * 0.9035
```

2.5.2.6 Faire des calculs statistiques avec la fonction *summarise()*

On calcule le prix moyen en fonction de la couleur et de la coupe du diamant :

```
calcul2 <- summarise(group_by(diamonds, cut, code_color), prix.moy = mean(price))
```

```
## `summarise()` has grouped output by 'cut'. You can override using the `~.groups` argument.
head(calcul2, 3)
```

```
## # A tibble: 3 x 3
## # Groups:   cut [1]
##   cut      code_color prix.moy
##   <ord>    <ord>      <dbl>
## 1 Fair     D          4291.
## 2 Fair     E          3682.
## 3 Fair     F          3827.
```

```
class(calcul2)
```

```
## [1] "grouped_df" "tbl_df"       "tbl"          "data.frame"
```

Sans le package **dplyr**, on aurait fait :

```
calcul2 <- aggregate(data.frame(price = diamonds[, "price"]),
                      list(color = diamonds$code_color,
                           cut = diamonds$cut), mean)
```

2.5.2.7 Faire des calculs statistiques en utilisant une nouvelle syntaxe

On souhaite calculer le prix maximum d'un diamant en fonction de la couleur et de la coupe. On ne veut garder que les prix supérieurs à 5000 dollars. Pour cela, on peut utiliser la syntaxe suivante, dite “pipelining”, qui provient du package **magrittr** (et importé par défaut via la package **dplyr**) :

```
diamonds %>%
  group_by(code_color, cut) %>%
  summarise(prix_groupe = mean(price)) %>%
  filter(prix_groupe > 5000)

## `summarise()` has grouped output by 'code_color'. You can override using the `.`group_by() function.

## # A tibble: 7 x 3
## # Groups:   code_color [3]
##   code_color cut      prix_groupe
##   <ord>       <ord>      <dbl>
## 1 H          Fair     5136.
## 2 H          Premium  5217.
## 3 I          Good    5079.
## 4 I          Very Good 5256.
## 5 I          Premium  5946.
## 6 J          Very Good 5104.
## 7 J          Premium  6295.
```

L'idée de cette syntaxe est de pouvoir comprendre rapidement les différentes opérations qui sont effectuées les unes à la suite des autres. Dans l'exemple précédent :

- on prend le jeu de données **diamonds**, le fait d'ajouter un “pipe” à la suite implique qu'on va appliquer une première opération dessus,
- *group_by* : on regroupe des observations en fonction des variables **color** et **cut**,
- *summarise* : on calcule le prix moyen sur les groupes précédemment créés,
- *filter* : on ne garde que les observations supérieures à 5000 dollars.

Remarque : depuis la version **4.1.0**, R a lancé une version native de l'opérateur ‘pipe’’. Il s'agit de la commande suivante|>‘. Voici un exemple d'utilisation :

```
my_vec <- rnorm(10)
my_vec |>
  mean()
```

```
## [1] -0.4518699
```

Dans la plupart des cas, il peut remplacer le pipe de **magrittr**.

2.5.2.8 Tirer un échantillon d'une population

On souhaite tirer de façon aléatoire 100 observations :

```
tirage1 <- sample_n(diamonds, 100)
```

On souhaite tirer de façon aléatoire 5% de l'échantillon :

```
tirage1 <- sample_frac(diamonds, 0.05)
```

2.6 Tidy data

Pour illustrer ce paragraphe, on considère les données suivantes : on observe pour 3 pays (Afghanistan, Brazil et Chine) sur 2 années consécutives (1999 et 2000), la taille de la population ainsi que le nombre de cas de tuberculoses observés. On va voir qu'on peut utiliser différentes tables pour présenter ces données et parmi ces différentes possibilités, on aura intérêt à en utiliser une plutôt que les autres. Pour charger ces données, on va installer le package **DSR** accessible depuis **github**. En effet, il est possible de récupérer sur **github** des packages qui sont en cours de développement et qui n'ont pas encore passés le processus de validation pour être un package officiel du CRAN :

```
devtools::install_github("garrettgman/DSR")
```

On pourrait traduire *tidy data* par données rangées en opposition à *messy data*, données désordonnées. Ce courant de *tidy data* vient encore de Hadley Wickham (voir article <https://www.jstatsoft.org/article/view/v059i10> ou encore <https://r4ds.had.co.nz/tidy-data.html>) qui part du principe suivant qui peut paraître évident, mais selon les situations, ce n'est pas toujours le cas :

- Une variable doit être rangée dans une colonne,
- Un individu est rangé dans une ligne,
- On place les valeurs observées dans les bonnes cases.

On présente ici une façon de présenter les données *tidy* :

```
DSR::table1
```

```
## # A tibble: 6 x 4
##   country     year   cases population
##   <fct>      <int>   <int>      <int>
## 1 Afghanistan 1999     745  19987071
## 2 Afghanistan 2000    2666  20595360
## 3 Brazil       1999  37737  172006362
```

```
## 4 Brazil      2000 80488 174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583
```

Au contraire, dans les données *messy*, on trouve en général une de ces situations :

- le nom des colonnes sont des valeurs et pas des noms,
- Plusieurs variables sont stockées dans une même colonne, c'est le cas du jeu de données suivant.

```
DSR::table3
```

```
## # A tibble: 6 x 3
##   country     year    rate
##   <fct>     <int> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil      1999 37737/172006362
## 4 Brazil      2000 80488/174504898
## 5 China       1999 212258/1272915272
## 6 China       2000 213766/1280428583
```

- Les variables sont à la fois présentes dans les lignes et les colonnes, c'est le cas du jeu de données suivant où constate que la colonne **key** contient le nom des deux variables **population** et **cases**, qu'il semblerait judiciable de présenter plutôt en colonnes.

```
DSR::table2
```

```
## # A tibble: 12 x 4
##   country     year key        value
##   <fct>     <int> <fct>     <int>
## 1 Afghanistan 1999 cases      745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000 cases      2666
## 4 Afghanistan 2000 population 20595360
## 5 Brazil      1999 cases      37737
## 6 Brazil      1999 population 172006362
## 7 Brazil      2000 cases      80488
## 8 Brazil      2000 population 174504898
## 9 China       1999 cases      212258
## 10 China      1999 population 1272915272
## 11 China      2000 cases      213766
## 12 China      2000 population 1280428583
```

- Les valeurs sont stockées dans plusieurs tables. C'est le cas de ces deux tables (une contenant les informations sur la population, l'autre sur le nombre de cas de tuberculoses) qui auraient pu être regroupées en une

seule table

```
DSR::table4
```

```
## # A tibble: 3 x 3
##   country    `1999` `2000`
##   <fct>      <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil        37737  80488
## 3 China         212258 213766
```

```
DSR::table5
```

```
## # A tibble: 3 x 3
##   country    `1999`    `2000`
##   <fct>      <int>     <int>
## 1 Afghanistan 19987071 20595360
## 2 Brazil      172006362 174504898
## 3 China       1272915272 1280428583
```

On va présenter ici les fonctions principales du package **tidyverse** (ce package fait partie de la bibliothèque **tidyverse**) qui permettent de passer d'un format *messy* à un format *tidy* ou inversement.

2.6.1 La fonction *pivot_longer()* : transformer des colonnes en lignes

Cette fonction permet de représenter en 1 seul colonne (et d'ajouter une colonne correspondant à une variable qualitative où les modalités sont le nom des variables initiales), une variable contenue à l'origine dans plusieurs colonnes.

The diagram illustrates the transformation of a wide table into a long table. On the left, a wide table 'DSR::table4' is shown with four columns: X1, X2, X3, and X4. The rows have values A and B. An arrow points from this table to the right, indicating the transformation process. On the right, the resulting long table 'DSR::table5' is shown. It has three columns: X1, V1, and V2. The X1 column contains categories A and B. The V1 column contains the names of the original columns (X2, X3, X4). The V2 column contains the corresponding values (1, 0.1, 10 for category A; 2, 0.2, 20 for category B).

X1	V1	V2
A	X2	1
B	X2	2
A	X3	0.1
B	X3	0.2
A	X4	10
B	X4	20

L'argument **cols** indique les variables à transformer en ligne, l'argument **names_to** correspond au nom donné à la variable contenant les nouvelles modalités (le nom des variables) et l'argument **values_to** correspond au nom de la colonne contenant les valeurs qui ont été transformées de colonnes en lignes.

```
pivot_longer(DSR::table4,
             cols = c("1999", "2000"),
             names_to = "years",
             values_to = "cases")
```

```
## # A tibble: 6 x 3
##   country     years   cases
##   <fct>      <chr>   <int>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000    2666
## 3 Brazil      1999   37737
## 4 Brazil      2000   80488
## 5 China       1999  212258
## 6 China       2000  213766
```

Avant *pivot_longer()*, il était possible d'utiliser la fonction *gather()*:

```
gather(DSR::table4, "year", "cases", 2:3)
```

```
## # A tibble: 6 x 3
##   country     year   cases
##   <fct>      <chr>   <int>
## 1 Afghanistan 1999     745
## 2 Brazil      1999   37737
## 3 China       1999  212258
## 4 Afghanistan 2000     2666
## 5 Brazil      2000   80488
## 6 China       2000  213766
```

Avant toutes ces fonctions, il aurait fallu exécuter ce genre de commandes où l'opération mathématique consiste à transformer une matrice en un vecteur.

```
data.frame(
  country = rep(DSR::table4$country, times = 2),
  year = rep(c("1999", "2000"), each = nrow(DSR::table4)),
  cases = as.vector(as.matrix(DSR::table4[, c("1999", "2000")]))
)

##       country year   cases
## 1 Afghanistan 1999     745
## 2      Brazil 1999   37737
## 3      China 1999  212258
## 4 Afghanistan 2000     2666
## 5      Brazil 2000   80488
## 6      China 2000  213766
```

2.6.2 La fonction *pivot_wider()* : transformer des lignes en colonnes

Cette fonction permet de re-distribuer les valeurs d'une colonne qui contenait l'information de plusieurs variables, en plusieurs colonnes où chaque colonne correspond à une variable.

X1	V1	V2
A	X2	1
A	X3	0.1
A	X4	10
B	X2	2
B	X3	0.2
B	X4	20

X1	X2	X3	X4
A	1	0.1	10
B	2	0.2	20

L'argument **names_from** correspond au nom de la colonne qui contient le nom des variables, et l'argument **values_from** correspond au nom de la colonne qui contient les valeurs à re-distribuer :

```
pivot_wider(DSR::table2, names_from = key,
            values_from = value)
```

```
## # A tibble: 6 x 4
##   country     year  cases population
##   <fct>      <int> <int>     <int>
## 1 Afghanistan 1999    745 19987071
## 2 Afghanistan 2000   2666 20595360
## 3 Brazil       1999  37737 172006362
## 4 Brazil       2000  80488 174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Avant *pivot_wider()*, il était possible d'utiliser la fonction *spread()*

```
spread(DSR::table2, key, value)
```

```
## # A tibble: 6 x 4
##   country     year  cases population
##   <fct>      <int> <int>     <int>
## 1 Afghanistan 1999    745 19987071
## 2 Afghanistan 2000   2666 20595360
## 3 Brazil       1999  37737 172006362
## 4 Brazil       2000  80488 174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Avant la création de ces fonctions, il aurait fallu utiliser la fonction *split()* et *merge()*:

```
sp_DSR <- split(DSR::table2[, c(1, 2, 4)], DSR::table2[, 3])
names(sp_DSR$cases)[3] <- "cases"
names(sp_DSR$population)[3] <- "population"
merge(sp_DSR$cases, sp_DSR$population,
      by.x = c("country", "year"),
      by.y = c("country", "year"),)
```

```
##      country year  cases population
## 1 Afghanistan 1999    745 19987071
## 2 Afghanistan 2000   2666 20595360
## 3      Brazil 1999 37737 172006362
## 4      Brazil 2000  80488 174504898
## 5      China 1999 212258 1272915272
## 6      China 2000 213766 1280428583
```

2.6.2.1 La fonction `separate()`

Cette fonction permet de séparer une colonne en deux variables dès qu'elle détecte un caractère de séparation. Par exemple, pour spliter la colonne `rate` de la `table3`.

```
separate(table3, rate, into = c("cases", "population"))

## # A tibble: 6 x 4
##   country     year  cases population
##   <chr>       <int> <chr>    <chr>
## 1 Afghanistan 1999  745  19987071
## 2 Afghanistan 2000 2666  20595360
## 3 Brazil      1999 37737 172006362
## 4 Brazil      2000 80488 174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583
```

2.6.2.2 La fonction `unite()`

Cette fonction permet de faire l'opération inverse de `separate()`. Elle permet de concaténer deux variables. Le résultat est proche de celui de la fonction `paste()`.

```
unite(DSR::table1, col = "rate",
      cases, population, sep = "/")

## # A tibble: 6 x 3
##   country     year rate
##   <fct>       <int> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil      1999 37737/172006362
## 4 Brazil      2000 80488/174504898
## 5 China       1999 212258/1272915272
## 6 China       2000 213766/1280428583
```

2.6.2.3 La fonction `extract()`

Cette fonction permet de spliter une colonne en deux en précisant quelle chaîne de caractère est utilisée pour le split. Dans l'exemple ci-dessous, on splitte

une chaîne de caractère lorsqu'il y a un espace dans une chaîne de caractère. Pour cela, on utilise l'argument **regex=** qui indique une expression régulière "REGEX".

```
df_to_split <- data.frame(
  player = c("Lionel Messi", "Christiano Ronaldo", "Antoine Griezman"),
  prix = c(170, 100, 120))
extract(df_to_split,
  col = player,
  into = c("prenom", "nom"),
  regex = "^(.).*(.).*$")  
  
##   prenom nom  prix
## 1      L   M  170
## 2      C   R  100
## 3      A   G  120
```

Ceci aurait pu se faire en utilisant la fonction *strsplit()*.

2.6.2.4 La fonction *complete()*

Cette fonction permet d'ajouter des lignes dans le cas où une valeur serait manquante. Par exemple, si on considère le jeu de données suivant, on constate que la firme B n'a pas de valeurs de CA pour l'année 2009.

```
firms <- data.frame(
  firms = c("A", "A", "B"),
  years = c("2008", "2009", "2008"),
  CA = c(20000, 25000, 40000)
)
```

Pour changer cela, on utilise la fonction *complete()* de la façon suivante:

```
complete(firms, firms, years)
```

```
## # A tibble: 4 x 3
##   firms years    CA
##   <chr> <chr> <dbl>
## 1 A     2008  20000
## 2 A     2009  25000
## 3 B     2008  40000
## 4 B     2009    NA
```

Pour plus de documentation sur l'univers **tidyR**, on recommande la lecture de cette page écrite par Julien Barnier.

Exercice 1.9.

Q1 Découper en 3 variables (**ville**, **num**, **dep**), le vecteur suivant :

```
code_INSEE <- c("toulouse_31_HG", "lyon_69_Rhone", "marsei_13_PACA")
```

2.7 Gestion de données volumineuses

Ici, nous allons essentiellement parler de données “moyennement volumineuses”, à savoir des données qui prennent entre 1 et 2 Go de RAM, ce qui correspond très approximativement à des jeux de données contenant quelques millions de lignes et quelques dizaines de variables. Nous parlerons des packages qui permettent de traiter des bases de données plus volumineuses, sans rentrer dans les détails.

2.7.1 Optimiser la fonction *read.table()*

2.7.1.1 Astuce 1

Une première astuce concernant la gestion de données volumineuses consiste à mieux gérer l’importation des données. Regardons ce que cela donne avec un fichier contenant 500000 lignes et 3 colonnes.

- Commençons par générer des données :

```
n <- 500000 # à modifier selon la mémoire vive de votre machine
donnees_a_importer <- data.frame(chiffre = 1:n,
lettre = paste0("caract", 1:n),
date = sample(seq.Date(as.Date("2017-10-01"), by = "day", len = 100), n,
replace = T))
object.size(donnees_a_importer)

## 40001136 bytes
str(donnees_a_importer)

## 'data.frame': 500000 obs. of 3 variables:
## $ chiffre: int 1 2 3 4 5 6 7 8 9 10 ...
## $ lettre : chr "caract1" "caract2" "caract3" "caract4" ...
## $ date   : Date, format: "2017-12-04" "2017-10-01" ...
```

Ce jeu de données utilise environ 38Mo de mémoire vive ou RAM (pour un rappel sur les différents types de mémoire, voir ce tutoriel intéressant). Il contient 3 variables : la 1ère au format **integer** (un **integer** occupe moins de mémoires qu’un **double**), la seconde au format **factor**, le troisième au format **Date**.

- Exportons ces données dans un fichier avec la fonction *write.table()* :

```
write.table(donnees_a_importer, "fichier.txt", row.names = F)
```

On peut connaître la taille du fichier en mémoire disque :

```
file.info("fichier.txt")
```

On dispose ainsi d'un fichier appelé *fichier.txt* dans l'espace de travail, qui pèse environ 16Mo.

- Importons ces données avec la fonction *read.table()* en mesurant le temps pris pour accomplir cette action. On remarque que les 2ème et 3ème variables ont été stockées en tant que **factor** (parce que on lui a demandé en utilisant l'option **stringsAsFactors = TRUE**).

```
system.time(import1 <- read.table("fichier.txt", header = T,
                                    stringsAsFactors = TRUE))

## utilisateur      système     écoulé
##       2.746        0.027      2.779

str(import1)

## 'data.frame': 500000 obs. of  3 variables:
##   $ chiffre: int  1 2 3 4 5 6 7 8 9 10 ...
##   $ lettre : Factor w/ 500000 levels "caract1","caract10",...: 1 111112 222223 333334 444445 455...
##   $ date   : Factor w/ 100 levels "2017-10-01","2017-10-02",...: 65 1 56 95 64 100 21 62 15 34 ...
```

- Exécutons la même opération en précisant que les chaînes de caractères seront stockées sous forme de **character**.

```
system.time(import2 <- read.table("fichier.txt", header = T,
                                    stringsAsFactors = FALSE))

## utilisateur      système     écoulé
##       0.456        0.004      0.460

str(import2)

## 'data.frame': 500000 obs. of  3 variables:
##   $ chiffre: int  1 2 3 4 5 6 7 8 9 10 ...
##   $ lettre : chr  "caract1" "caract2" "caract3" "caract4" ...
##   $ date   : chr  "2017-12-04" "2017-10-01" "2017-11-25" "2018-01-03" ...
```

Le gain de temps dans le second cas vient du fait qu'on demande à ce que les variables qualitatives ne soient pas codées en **factor**. En effet, pour créer un **factor**, R a besoin de parcourir l'ensemble du fichier pour identifier tous les **levels** possibles, pour pouvoir attribuer un numéro à chaque **levels**. Cette procédure n'est évidemment pas optimale et il s'agissait d'une critique majeure concernant les objets **data.frame**, à savoir que les variables qualitatives sont codées par défaut en **factor** jusqu'à 2019. Depuis la version 4.0.0., l'argument ***stringsAsFactors**** vaut :

```
default.stringsAsFactors()

## [1] FALSE
```

En effet, depuis la conférence useR!2019 qui s'est tenue à Toulouse, les chaînes

de caractères sont à présent sauvegardées au format **character**. L'idée avancée était qu'un **factor** ordonne les caractères selon des règles lexicographiques qui ne sont pas les mêmes d'un pays à un autre. Autrement dit, compte tenu que ces règles sont définies par la machine utilisée, on peut obtenir des résultats différents d'une machine à une autre et la reproductibilité d'un même code n'était donc plus assurée. Pour plus de détails , voir ce message.

2.7.1.2 Astuce 2

Pour importer un gros fichier, nous conseillons d'utiliser dans un premier temps la fonction `read.table()` (ou `read.csv()`, etc) sur les quelques premières lignes du fichier (entre 20 et 100 lignes selon la taille du fichier), puis d'analyser la structure du jeu de données, plus particulièrement le type de chaque colonne :

```
bigfile_sample <- read.table("fichier.txt", stringsAsFactors = FALSE,
                             header = T, nrows = 20)
bigfile_colclass <- sapply(bigfile_sample, class)

##     chiffre      lettre      date
## "integer" "character" "character"
```

Dans un second temps, on importe la table en précisant le type de chaque colonne (par défaut l'option `stringsAsFactors = FALSE` donc ce n'est pas la peine de l'ajouter), ce qui aura pour effet de diminuer encore légèrement le temps de traitement.

```
system.time(bigfile_raw <- read.table("fichier.txt", header = T,
                                         colClasses = bigfile_colclass))

## utilisateur      système      écoulé
##       0.338        0.003        0.343
```

Remarque 1 : si le type d'une colonne a été mal spécifié, cela pourra créer un message d'erreur.

Remarque 2 : si vous êtes sûr que le fichier ne contient pas de lignes de commentaires, vous pouvez encore améliorer le temps de lecture en précisant l'option `comment.char=""`, ce qui permettra d'éviter de faire une recherche systématique de caractères de commentaires.

2.7.2 Le package `readr`

Nous présentons ici le package **readr**, faisant également partie du projet **tidyverse** et dont l'objectif est encore et toujours de rendre les codes plus simples et calculs plus rapides. Pour cela, les fonctions de ce package utilisent du code **C++** ce qui vous le verrez permet de faire des améliorations considérables en temps de calcul. Parmi les fonctions de ce package, `read_csv()` ou `read_table()` dont le but est bien entendu l'importation de fichiers **csv** ou **txt**. Elles ont

étés programmées de telle sorte qu'il suffit en général de les appeler en précisant uniquement le chemin d'accès du fichier à importer.

```
system.time(
  tibble.don <- read_table2("fichier.txt")
)

## 
## -- Column specification -----
## cols(
##   `chiffre` = col_double(),
##   `lettre` = col_character(),
##   `date` = col_date(format = "")
## )

## utilisateur      système      écoulé
##       0.324        0.004        0.330
```

L'objet importé n'est pas un **data.frame** mais un **tibble** que nous avons déjà vu précédemment :

```
class(tibble.don)

## [1] "spec_tbl_df" "tbl_df"        "tbl"           "data.frame"
object.size(tibble.don)

## 44004504 bytes
```

Ce type de format ne s'est pas encore généralisé à toutes les fonctions de **R**. Il se peut donc que vous rencontriez des difficultés pour utiliser certaines fonctions sur ce type d'objets. Pour passer du format **tibble** au **data.frame**, cela se fait très facilement au moyen de la fonction **as.data.frame**.

```
don <- as.data.frame(tibble.don)
```

Autres formats de données : dans la même lignée que **readr**, nous citons également le package **readxl** pour la lecture de fichiers *xls/xlsx*, ainsi que le package **haven** pour la lecture de données issue des logiciels **SPSS**, **Stata** ou **SAS**.

Bibliographie: pour l'usage du package **readr**, le lecteur pourra consulter ce document <http://readr.tidyverse.org/>.

2.7.3 Autres packages

Voici 3 packages, parmi d'autres, susceptibles d'aider l'utilisateur à manipuler des fichiers de données volumineux.

2.7.3.1 Package data.table

Ce package est un package concurrent au projet **tidyverse**, l'objectif de ce package étant également de rendre les lignes de codes plus élégantes, les calculs plus rapides, notamment en présence de données volumineuses. Le package **data.table** est plus ancien que le projet **tidyverse**, mais ce dernier bénéficiant du support financier de **RStudio**, il semble à ce jour plus intéressant de choisir l'univers **tidyverse** qui devrait bénéficier de plus de développements par la suite. Nous présentons toutefois ici quelques exemples d'utilisation de **data.table**.

Pour importer un jeu de données :

```
require("data.table")
system.time(
  objet.data.table <- fread("fichier.txt")
)

## utilisateur      système      écoulé
##       0.411        0.004       0.091
```

L'objet créé appartient à la classe d'objet **data.table** :

```
class(objet.data.table)

## [1] "data.table" "data.frame"
object.size(objet.data.table)

## 40001760 bytes
```

Pour manipuler ce format de données, il faut se familiariser avec une nouvelle syntaxe propre à ce genre de données. Le lecteur pourra consulter la note suivante s'il souhaite utiliser ce package : <https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html>

2.7.3.2 Package ff

Pour des données trop volumineuses par rapport à la mémoire vive disponible de la machine, la package **ff** va faire en sorte de ne charger qu'une partie des données en mémoire vive que lorsqu'il en aura besoin.

Ici, on crée un fichier “*big_file.csv*” qui va contenir 50,000,000 observations et 3 colonnes. Pour faire cela, on va concaténer 100 fois **Donnees.a.importer** au fichier de sortie *big_file.csv* en utilisant la fonction *write_csv()*. L'opération peut prendre quelques minutes et c'est pourquoi pour informer l'utilisateur du temps restant, on propose d'utiliser la fonction *progress_estimated()*, qui fait avancer une barre au début de chaque nouvelle boucle (ici la boucle est répétée 100 fois).

```
write_csv(donnees_a_importer, "big_file.csv")
```

```
p <- progress_estimated(100)
for(k in 1:100){
  p$pause(0.1)$tick()$print()
  write_csv(donnees_a_importer, "big_file.csv", append = T)
}
```

La taille du fichier créé, d'environ 1.5Go est encore théoriquement manipulable sur **R** (essayer de le faire via la fonction `read.csv()`), mais on va supposer ici qu'on ne souhaite pas charger ce jeu de données intégralement en mémoire vive. Pour cela, on va utiliser la fonction `read.csv.ffdf()` du package **ff**. On spécifie comme options que l'on souhaite lire les données par groupe de 5,000,000 d'observations, excepté la première fois où l'on va lire 500,000 observations

```
require("ff")
bigDF <- read.csv.ffdf(file="big_file.csv", header = TRUE,
                        first.rows = 500000, next.rows = 5000000)
```

L'objet créé ne prend qu'une partie de la mémoire vive.

```
bigDF
object.size(bigDF)
```

L'idée de ce package est de stocker les variables non pas en mémoire vive, mais quelque part sur le disque dur et d'y accéder en utilisant des pointeurs. On peut effectuer un certain nombre d'opérations sur ces objets. Par exemple, pour calculer la moyenne de la variable **chiffre** :

```
library("ffbase")
mean.ff(bigDF$chiffre)
```

Remarque: on citera également le package **bigmemory** dont le principe est similaire. Plus récemment, le package **disk.frame** semble également promis à un bel avenir.

2.7.4 Algorithme de type Map/Reduce

On peut utiliser un algorithme de type *Map/Reduce* pour éviter d'importer un jeu de données trop volumineux sur la RAM. A la base, ce type d'algorithme s'applique sur des données qui sont organisées selon l'approche conceptuelle d'Hadoop, où les fichiers de données sont fractionnés en gros bloc et distribués à travers les noeuds d'un cluster.

Dans cet exemple, nous n'avons qu'un gros fichier de données, l'idée étant d'importer des échantillons de ce jeu de données sur lesquels on va appliquer la première étape *Map* de l'algorithme.

Par exemple, dans l'exemple ci-après, on a choisi d'importer les 5,000,000 premières observations, puis les 5,000,000 suivantes, etc. jusqu'à ce qu'on est parcouru tout le fichier. Sur chacun de ces échantillons, on va calculer d'une part

la somme et d'autre part le maximum d'une variable quantitative.

Une fois réalisée cette étape sur les sous-échantillons, on va assembler les résultats obtenus à l'étape précédente. Pour calculer la moyenne, on va faire la somme sur les sommes obtenues et diviser ensuite par le nombre total d'observations et pour le max, on va calculer le maximum sur les maximum.

Remarque : on ne peut pas appliquer toutes les méthodes statistiques sur ce type d'algorithme (par exemple calculer un quantile nécessite de travailler sur l'échantillon complet). En revanche, ce type d'algorithme peut fonctionner pour calculer l'estimateur des moindres carrés d'un modèle linéaire. Par ailleurs, le calcul parallèle (que nous verrons dans un autre chapitre) pourra être envisagé sur ce type d'algorithme.

```
n_split <- 11
ind <- 1
n_max <- 5000001
my_max <- numeric(n_split)
my_mean <- numeric(n_split)
my_n <- numeric(n_split)
for (k in 1:n_split) {
  split_don <- read_csv("big_file.csv", skip = ind, n_max = n_max,
                        col_names = c("chiffre", "lettre", "date"),
                        col_types = "ncc")
  my_max[k] <- max(split_don$chiffre)
  my_mean[k] <- sum(split_don$chiffre)
  my_n[k] <- nrow(split_don)
  ind <- ind + n_max
}
sum(my_mean) / sum(my_n)
max(my_max)
```

2.7.4.1 Package Matrix

Il est possible d'avoir suffisamment de mémoires vives pour importer des données, mais pour certaines opérations algébriques et notamment le calcul matriciel, il y a des cas où on a besoin d'avoir plus de mémoires vives que ce dont nous disposons (typiquement une inversion de matrice). Dans ce cas-là, il est important de voir si les données sur lesquelles on travaille sont creuses ou non, c'est-à-dire contenant beaucoup de 0. Si c'est le cas, le package **Matrix** permet d'obtenir de bonnes performances en temps calcul et d'utiliser moins de RAM, grâce aux propriétés des matrices creuses.

Ce package s'utilise très simplement et la plupart des fonctions existantes pour le calcul matriciel (*crossprod()*, *solve()*, *%>%*) s'appliquent directement sur ce type d'objet. Pour plus d'informations sur ce package, consulter la vignette.

```
library("Matrix")
mat <- matrix(rbinom(10000, 1, 0.05), 100, 100)
object.size(mat)
Mat <- as(mat, "Matrix")
object.size(Mat)
```

2.7.5 Interaction avec des systèmes de gestion de bases de données

Lorsqu'une entreprise travaille sur de gros volumes de données, cela sous-entend qu'elle dispose de systèmes de gestion de base de données.

R dispose de plusieurs packages permettant d'intégrer avec des systèmes de gestion de base de données : **RODBC**, **RMySQL**, **RPostgreSQL**, **RSQLite** ainsi qu'avec des bases de données orientées document comme *MongoDB* et *couchDB* via les packages **mongolite** et **couchDB**.

Le gros avantage de ces packages est qu'ils permettent de laisser les bases de données trop grandes sur l'espace disque et d'aller récupérer uniquement l'information dont on a besoin en envoyant depuis **R** des requêtes qui seront exécutées sur les systèmes de gestion de base de données.

Bibliographie pour le traitement de données volumineuses, nous recommandons la lecture de ce document : https://rpubs.com/msundar/large_data_analysis

2.7.6 Utiliser la syntaxe SQL

Pour celles et ceux qui sont familiers avec le langage SQL, le package **sqldf** permet d'utiliser la syntaxe SQL pour faire des requêtes depuis **R**.

Pour plus d'informations, voir : <https://github.com/ggrothendieck/sqldf>

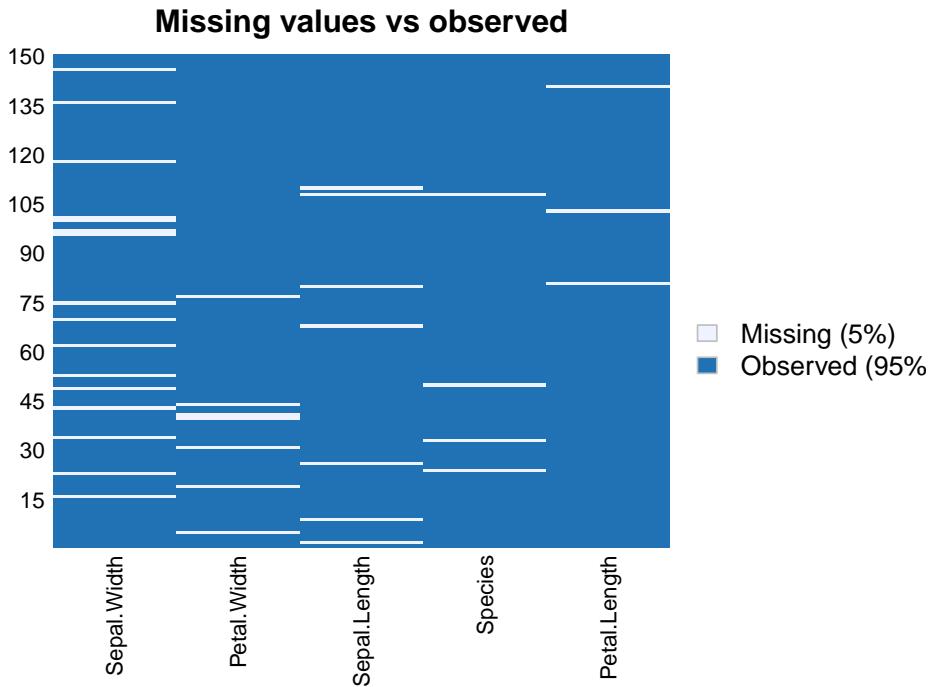
2.8 Visualiser et traiter les données manquantes

On présente dans ce paragraphe quelques packages qui permettent de visualiser et traiter les données manquantes. Pour commencer, nous allons générer des valeurs manquantes de façon aléatoire dans le jeu de données **iris** :

```
require("missForest")
iris.mis <- prodNA(iris, 0.05)
```

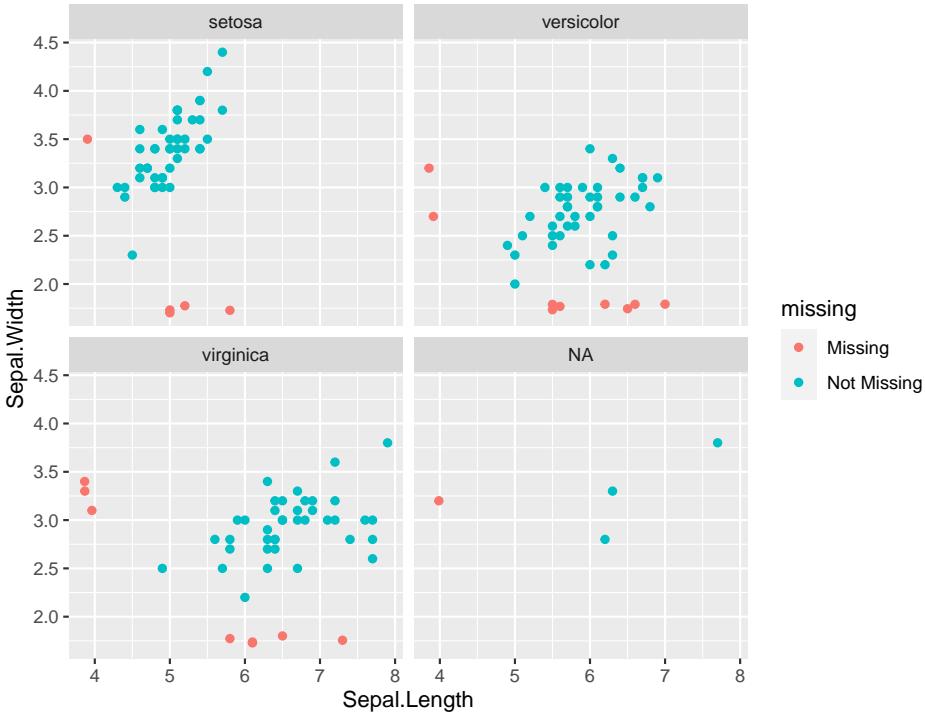
Le package **Amelia** permet de visualiser dans un graphique où sont localisées les données manquantes. Cela permet notamment de voir s'il existe un pattern ou une forme particulière (un bloc par exemple) parmi les valeurs manquantes :

```
require("Amelia")
missmap(iris.mis, main = "Missing values vs observed")
```



On notera également les packages **visdat** et **naniar** qui proposent plusieurs outils pour visualiser les données manquantes. Par exemple, pour savoir si les valeurs manquantes d'une variable sont liées à une autre variable, on peut utiliser l'outils suivant :

```
ggplot(iris.mis, aes(x = Sepal.Length, y = Sepal.Width)) +
  naniar::geom_miss_point() +
  facet_wrap(~Species)
```



Pour traiter les données manquantes, il existe plusieurs façons de procéder :

- ne rien faire. Dans ce cas, certaines fonctions comme la fonction `lm()` enlève automatiquement les observations dès lors qu'il existe au moins une valeur manquante parmi les variables à expliquer et explicatives :

```
res_lm <- lm(Sepal.Width ~ Sepal.Length + Petal.Length +
  Petal.Width + Species, data = iris.mis)
```

- suppression des observations contenant au moins une valeur manquante en utilisant la fonction `subset()` ou `filter()` (du package `dplyr`)

```
iris.mis <- subset(iris.mis, !is.na(Species))
```

- imputation par la moyenne ou la médiane lorsqu'il s'agit d'une variable numérique ou alors par le mode lorsqu'il s'agit d'une variable qualitative.

```
ind_NA_Sepal.Length <- is.na(iris.mis$Sepal.Length)
mean_spec <- aggregate(Sepal.Length ~ Species,
  data = iris.mis, FUN = mean, na.rm = T)
for (k in levels(iris.mis$Species)) {
  iris.mis[ind_NA_Sepal.Length & iris.mis$Species == k,
    "Sepal.Length"] <- mean_spec[mean_spec$Species == k, 2]
}
```

- imputation en utilisant des modèles linéaires afin de prédire les valeurs manquantes. Par exemple :

```
iris.imp_mean <- data.frame(sapply(iris.mis[, 1:4],
  function(x) ifelse(!is.na(x), x, mean(x, na.rm = T))),
  Species = iris.mis$Species)
pred <- predict.lm(res_lm, newdata = iris.imp_mean)
iris.mis[is.na(iris.mis$Sepal.Width), "Sepal.Width"] <-
  pred[is.na(iris.mis$Sepal.Width)]
```

- imputation en utilisant des modèles de prédiction sophistiqués issus du machine learning. Par exemple, les forêts aléatoires ou les K -plus proches voisins :

```
require("missForest")
iris.imp <- missForest(iris.mis)

## missForest iteration 1 in progress...done!
## missForest iteration 2 in progress...done!
## missForest iteration 3 in progress...done!
## missForest iteration 4 in progress...done!

require("VIM")
iris.knn <- kNN(iris.mis, k = 2)
```

2.9 Répertoires et fichiers

Il existe plusieurs fonctions de base qui permettent la gestion des répertoires et fichiers. L'utilisation de la plupart de ces fonctions est intéressante lors de l'écriture de scripts “propres” qui stockeront les résultats dans des fichiers bien rangés dans des répertoires bien nommés.

Parmi ces fonctions :

- la fonction `getwd()` retourne le chemin du répertoire de travail. Il s'agit du répertoire où seront sauvegardées par défaut les différentes opérations de sauvegarde (graphiques, codes, fichier historique, etc.) :

```
getwd()
```

```
## [1] "/home/laurent/Documents/R_book/R_book"
```

- la fonction `dir()` retourne les fichiers et répertoires situés dans le chemin spécifié et éventuellement ses sous-répertoires (options `recursive = TRUE`). Par défaut, il s'agit du chemin correspondant à l'environnement de travail actuel (donné par `getwd()`) :

```
dir()
```

```
## [1] "_book"                      "_bookdown_files"           "_bookdown.yml"
```

```
## [4] "_main_files"           "_main.log"           "_main.pdf"
## [7] "_main.Rmd"              "_main.tex"             "_output.yml"
## [10] "01-data-management.Rmd" "02-programming.Rmd"   "03-parallel.Rmd"
## [13] "04_graphics.Rmd"        "05-intro.Rmd"         "06-cross-refs.Rmd"
## [16] "07-parts.Rmd"            "07-references.Rmd"  "08-citations.Rmd"
## [19] "09-blocks.Rmd"           "10-share.Rmd"          "book.bib"
## [22] "fichier.txt"            "Figures"             "index.Rmd"
## [25] "packages.bib"            "preamble.tex"          "R"
## [28] "R_book.Rproj"            "README.md"            "Sorties"
## [31] "style.css"
```

- la fonction *file.info()* prend comme argument d'entrée des chemins de fichiers ou répertoires et retourne des informations les concernant (taille, etc.) :

```
file.info(dir())
```

- la fonction *R.home()* retourne l'emplacement de **R** :

```
R.home()
```

```
## [1] "/usr/lib/R"
```

- la fonction *file.access()* donne des informations sur la permission accordée aux fichiers. Les permissions qui sont testées sont : existence (0), exécution (1), écriture (2) et lecture (4). La fonction retourne la valeur 0 pour oui et -1 pour non. Dans l'exemple qui suit, les fichiers qui sont situés dans le chemin où se situe **R** peuvent être lus et exécutés, mais sont en revanche protégés en écriture :

```
fic <- dir(file.path(R.home(), "bin"), full = T)
file.access(fic, 0)
file.access(fic, 1)
file.access(fic, 2)
file.access(fic, 4)
```

- la fonction *dir.exists()* teste l'existence d'un répertoire et la fonction *dir.create()* crée un répertoire.

```
dir.create("Sorties")
```

```
## Warning in dir.create("Sorties"): 'Sorties' existe déjà
dir.exists("Sorties")
```

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

Exercice 1.10.

écrire une fonction qui, à partir d'un nombre entier n passé en paramètre, effectue un tirage aléatoire de n valeurs selon une loi normale $\mathcal{N}(0, 1)$ puis renvoie

les valeurs générées dans un fichier d'un répertoire *Num* et trace une boxplot de ces données qui sera stockée dans un fichier *.jpg* du répertoire *Graph*.

2.10 Autour de l'espace de travail

La fonction *search()* donne la liste des packages (mais pas seulement) attachés à l'espace de travail. En gros, il s'agit des packages et environnements auxquels il est possible d'accéder dans la session en cours à l'instant *t*. Cette liste évolue bien entendu au fur et à mesure de la session :

```
search()
library("foreign")
search()
```

L'option **pos** de la fonction *ls()* permet de lister le contenu d'un package particulier en donnant sa position dans la liste renvoyée par *search()*. Vous pourrez en choisissant le bon indice, vous rendre compte de l'ensemble des fonctions accessibles depuis le package **base** :

```
ls(pos = which(search() == "package:base"))
```

Certains packages peuvent avoir des fonctions portant le même nom ; d'où le message d'avertissement ci-dessous au moment de charger un nouveau package, indiquant qu'un objet est masqué dans un package situé plus loin dans la liste *search()*.

```
library("pls")

##
## Attachement du package : 'pls'
## L'objet suivant est masqué depuis 'package:stats':
##   loadings
search()

## [1] ".GlobalEnv"           "package:pls"          "package:VIM"
## [4] "package:grid"          "package:colorspace"    "package:Amelia"
## [7] "package:Rcpp"           "package:missForest"    "package:itertools"
## [10] "package: iterators"     "package:foreach"       "package:randomForest"
## [13] "package: data.table"    "package:forcats"       "package:dplyr"
## [16] "package: purrr"         "package:readr"         "package:tidyverse"
## [19] "package: tibble"        "package:ggplot2"       "package:tidyverse"
## [22] "package: gplots"        "package:zoo"           "package:classInt"
## [25] "package: glue"          "package:stringr"      "package:wordcloud"
## [28] "package: RColorBrewer"   "package:stats"         "package:graphics"
## [31] "package: grDevices"     "package:utils"         "package:datasets"
## [34] "package: methods"       "Autoloads"            "package:base"
```

Dans ce cas, si on souhaite obtenir l'aide de la “bonne” fonction, il suffit de précéder le nom de la fonction par le nom du package suivi de ::.

```
find("loadings")
## [1] "package:pls"    "package:stats"
?loadings

## Help on topic 'loadings' was found in the following packages:
##
##   Package           Library
##   pls                /home/laurent/R/x86_64-pc-linux-gnu-library/4.1
##   stats              /usr/lib/R/library
##
## 
## Utilisation de la première correspondance ...
?stats::loadings
```

Remarque : on pourra procéder de la même façon pour être sûr d'exécuter la “bonne” fonction.

On a vu précédemment que pour accéder à un élément d'une **list** ou d'un **data.frame**, il est nécessaire d'appeler cet objet suivi de l'opérateur **\$**. Il existe au moins deux façons de simplifier cette opération. La première consiste à utiliser la fonction **with()** :

```
with(airquality,
     summary(Ozone))

##   Min. 1st Qu. Median   Mean 3rd Qu.   Max.   NA's
##   1.00  18.00  31.50  42.13  63.25 168.00      37
```

La seconde consiste à utiliser la fonction **attach()** qui permet d'attacher tous les éléments d'un objet dans l'environnement de travail comme s'il s'agissait d'un package.

```
e <- list(e_vec_x = rnorm(10),
          e_fic1 = function(x) mean(x),
          e_fic2 = function(x) mean((x-e_fic1(x))^2))
attach(e)
search()

## [1] ".GlobalEnv"        "e"                  "package:pls"
## [4] "package:VIM"         "package:grid"        "package:colorspace"
## [7] "package:Amelia"       "package:Rcpp"         "package:missForest"
## [10] "package:itertools"     "package:iterators"   "package:foreach"
## [13] "package:randomForest"  "package:data.table"   "package:forcats"
## [16] "package:dplyr"        "package:purrr"        "package:readr"
```

```
## [19] "package:tidyverse"      "package:tibble"        "package:ggplot2"
## [22] "package:tidyverse"      "package:gplots"        "package:zoo"
## [25] "package:classInt"       "package:glue"          "package:stringr"
## [28] "package:wordcloud"       "package:RColorBrewer"   "package:stats"
## [31] "package:graphics"        "package:grDevices"     "package:utils"
## [34] "package:datasets"        "package:methods"       "Autoloads"
## [37] "package:base"
```

Ensuite, il est possible d'accéder directement aux éléments de la liste sans avoir à appeler l'objet **e** :

```
e_fic1(e_vec_x)
## [1] 0.6690173
e_fic2(e_vec_x)
## [1] 0.93999229
detach(e)
```

Attention : il faut être très prudent avec la fonction *attach()*, car cela peut créer des conflits avec l'environnement global.

Remarque : il est parfois utile de connaître certains détails de l'environnement de travail notamment pour comprendre *pourquoi ça ne marche pas !!!* (voir dessin ci-dessous). Les fonctions *sessionInfo()* et *R.Version()* peuvent permettre d'identifier certaines incompatibilités entre notre version de **R** et un package particulier que l'on vient d'installer.



Figure 2.1: ça ne marche pas !!!

Exercice 1.11.

Q1 Quel est l'inconvénient illustré par les manipulations de *e_fic1()*, **e_vec_x** et *e_fic2()* ?

Q2 à quoi sert l'opérateur `::` ?

Chapter 3

Programmation

Nous allons voir dans ce chapitre des éléments de programmation qui permettent de gagner en clarté et simplicité dans le code et parfois en temps de calcul.

Ce document a été généré directement depuis **RStudio** en utilisant l'outil Markdown. La version *.pdf* se trouve ici.

Packages à installer

```
install.packages((c("ggplot2", # Graphiques ggplot2
                    "kableExtra", # Insérer des tables dans Markdown
                    "Matrix", # Matrice creuse
                    "microbenchmark", # Temps de calcul
                    "pryr", # Mieux comprendre R
                    "Rcpp", # Faire appel à du code C++
                    "reticulate" # Interface vers Python
)))
```

On montre un exemple pour utiliser du code **Python**. Pour cela, on aura besoin de la bibliothèque **pandas**:

```
reticulate::py_install("pandas")
```

3.1 Quelques règles de style

R n'a pas de PEP 8 comme c'est le cas pour Python. En revanche, pour des raisons évidentes de clarté et de lisibilité pour soi-même et pour les éventuels collègues qui vont lire notre code, il est important d'essayer de respecter si possible quelques règles d'écritures de codes. On s'est inspiré ici de ces deux documents :

- “Style guide” de Hadley Wickham dans le livre “**R** advanced”

- “Google’s R style guide”

3.1.1 Le nom des fichiers de codes

Appeler vos fichiers de code avec l’extension `.R` et donner des noms clairs à vos fichiers de codes, en utilisant éventuellement le symbole underscore ou le trait d’union entre les différents mots. Par exemple :

```
traitement-base.R
fonctions-utiles.R
exploration.R
prediction.R
```

3.1.2 Le nom des objets

Eviter d’appeler vos objets avec des noms de fonctions existants ou de mots clés. Le nombre de caractères devrait être restreint même si le nom doit avoir un sens pour le lecteur. On peut utiliser le symbole underscore ou le point pour séparer des mots entre eux :

```
arbre_reg
rf_reg
glm_reg
```

3.1.3 Espace entre les opérateurs

Aérer au maximum les opérations (affectation, calcul, utilisation de fonctions, etc.) par des espaces :

```
a <- c(5, NA, 4, 3)
mean(a, na.rm = TRUE)
```

Exception faite pour les opérateurs `:`, `::` et `:::` où il ne faut pas d’espace :

```
1:10
stats::lm
```

3.1.4 Les conditions

Quand on utilise les conditions `if/else`, `for`, `while` :

- laisser un espace après le mot clé,
- l’accolade ouvrante se trouve à la fin de la ligne contenant le mot clé, l’accolande fermante se trouve sur une nouvelle ligne,
- mettre deux espaces en début de ligne (indentation) à partir de la seconde ligne jusqu’à la fin de la condition,

- si des conditions sont imbriquées, la nouvelle condition devrait se trouver sur la même ligne que l'accolade fermante de la première condition.

```
if (y < 0 && debug) {
  message("Y is negative")
}

if (y == 0) {
  log(x)
} else {
  y ^ x
}
```

3.1.5 La taille d'une ligne

Essayer de limiter 80 caractères à une ligne de code.

3.1.6 Affectation

Pour affecter une valeur à un objet essayer d'utiliser l'opérateur `<-` plutôt que `=`.

```
a <- 5
```

Remarque : sur **RStudio** quelques-une des règles ci-dessus sont implémentées par défaut, notamment les indentations après les boucles ou les fonctions.

Exercice 2.1

Mettre le code suivant en utilisant les règles de style présentées ci-dessus:

```
my_mean=function(x)
{
  # verification
  if(!is.numeric(x))
    stop("x must be a numeric vector")
  # initialization
  n= length(x)
  res =0
  for(k in 1:n)
  {
    res= res+x[k]
  }
  # return res
  return(res/ n)
}
```

3.2 Fixer la taille des vecteurs (si on la connaît à l'avance)

On a le problème suivant : on souhaite obtenir un vecteur de taille n contenant des valeurs simulées issues d'une loi gaussienne centrée et réduite, mais on ne souhaite garder que les valeurs supérieures à un paramètre a (égal à 0 par défaut). Dans la fonction `trunc_rnorm.1()` ci-dessous, nous ne précisons pas la taille du vecteur \mathbf{x} qui sera retourné. A chaque fois qu'une valeur répond au critère, on la concatène au vecteur \mathbf{x} en utilisant la commande `c()`. Le défaut de cette méthode est que **R** est sans arrêt en train d'allouer un nouvel espace mémoire pour le vecteur \mathbf{x} qu'on modifie à chaque fois puisqu'on change sa taille.

```
trunc_rnorm.1 <- function(n, a = 0) {
  x <- numeric(0)
  i <- 1
  while (i <= n) {
    r <- rnorm(1)
    if (r > a) {
      x <- c(x, r)
      i = i + 1
    }
  }
  x
}
```

C'est pourquoi lorsqu'on connaît à l'avance la taille du vecteur qu'on souhaite créer, on crée le vecteur avec la bonne taille et l'espace mémoire alloué à ce vecteur est fixée dès le début. C'est la seule différence avec la fonction `trunc_rnorm.2()` ci-dessous où on a précisée la taille de \mathbf{x} dès le départ. On affecte ensuite à chaque élément de \mathbf{x} la valeur qu'on souhaite garder.

```
trunc_rnorm.2 <- function(n, a = 0) {
  x <- numeric(n)
  i <- 1
  while (i <= n) {
    r <- rnorm(1)
    if (r > a) {
      x[i] <- r
      i <- i + 1
    }
  }
  x
}
```

On constate que les différences de temps de calcul sont assez importantes :

```
system.time(trunc_rnorm.1(10000))
```

## utilisateur	système	écoulé
----------------	---------	--------

```

##          0.441      0.024      0.465
system.time(trunc_rnorm.2(10000))

## utilisateur     système     écoulé
##          0.055      0.000      0.055

```

3.3 Temps de calcul et mémoire

3.3.1 Pour mesurer les temps de calcul efficacement

Si vous répétez les instructions précédentes successivement, vous constaterez que les temps de calculs sont à chaque fois différents. Ceci s'explique par le fait que les machines sur lesquelles on travaille exécutent plusieurs tâches à la fois, qui ne sont pas toujours les mêmes à l'instant t . Aussi, une façon de rendre robuste ces temps de calcul est de répéter un certain nombre de fois ces mêmes commandes et de présenter ensuite les résultats statistiques sur ces temps de calcul.

Pour cela, on peut bien évidemment utiliser une boucle **for** ainsi et faire ensuite des statistiques de base

```

my_time <- data.frame(time_1 = numeric(10),
                       time_2 = numeric(10))
for (b in 1:10) {
  my_time$time_1[b] <- system.time(trunc_rnorm.1(10000))[3]
  my_time$time_2[b] <- system.time(trunc_rnorm.2(10000))[3]
}
summary(my_time)

##      time_1          time_2
##  Min.   :0.4530   Min.   :0.04100
##  1st Qu.:0.4550   1st Qu.:0.04125
##  Median :0.4565   Median :0.04200
##  Mean   :0.4976   Mean   :0.04310
##  3rd Qu.:0.4895   3rd Qu.:0.04200
##  Max.   :0.7820   Max.   :0.05400

```

Sinon, on utilise la fonction *microbenchmark()* du package **microbenchmark** (Mersmann et al., 2019). En général, on regarde la moyenne.

```

mbm <- microbenchmark::microbenchmark(
  trunc_rnorm.1(n = 10000),
  trunc_rnorm.2(n = 10000),
  times = 10L)
mbm

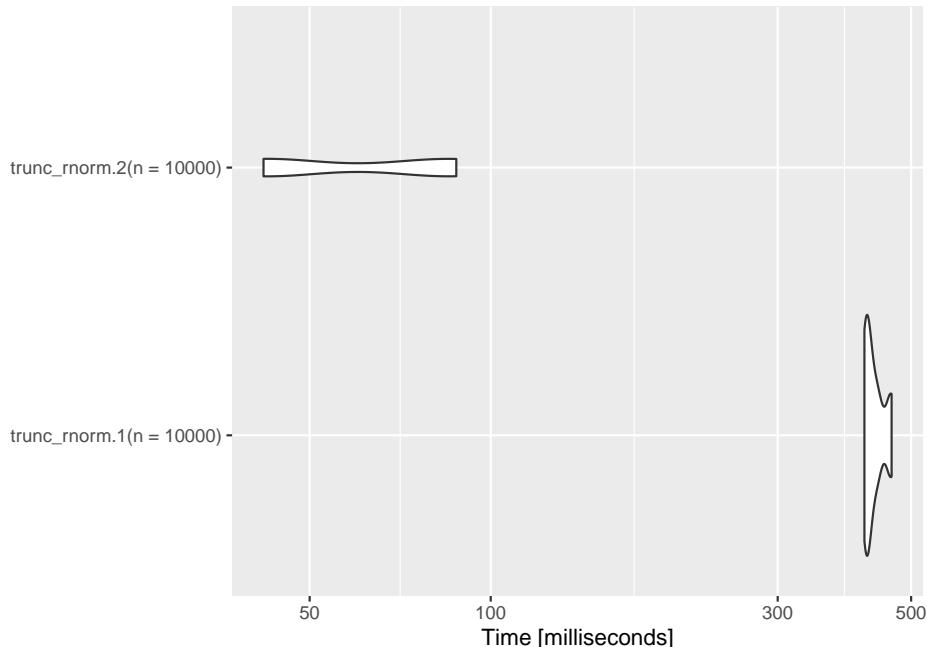
## Unit: milliseconds
##                                expr      min       lq      mean      median      uq

```

```
##  trunc_rnorm.1(n = 10000) 418.11913 422.05836 433.55421 424.85583 441.08282
##  trunc_rnorm.2(n = 10000) 41.90505 42.17059 63.89793 62.71539 85.59846
##          max  neval
##  463.94235     10
##  87.65357     10
```

Par ailleurs, il est possible de représenter graphiquement ces résultats avec la fonction *autoplot()* du package **ggplot2**.

```
ggplot2::autoplot(mbm)
```



3.3.2 Pour comprendre la gestion de la mémoire

Lorsqu'on crée un vecteur sous **R**, celui-ci est stocké sur un espace mémoire que l'on peut identifier avec la fonction *address()* du package **pryr** (Wickham, 2019).

Par exemple :

```
library("pryr")
```

```
## 
## Attachement du package : 'pryr'
## L'objet suivant est masqué depuis 'package:data.table':
## 
##      address
```

```
## Les objets suivants sont masqués depuis 'package:purrr':
##
##     compose, partial
x <- numeric(10)
address(x)

## [1] "0x55cc53fca7a8"
```

Lorsqu'on attribue de nouvelles valeurs à ce vecteur, l'adresse ne change pas :

```
for (i in 1:10) {
  x[i] <- ifelse(rnorm(1) > 0, 1, 0)
  print(address(x))
}
```

```
## [1] "0x55cc53f04a68"
```

En revanche, si on change la taille de ce vecteur, R va systématiquement réserver un nouvel espace mémoire, ce qui est une opération coûteuse en temps calcul, d'où des temps de calcul plus long. Par exemple :

```
for (i in 11:20) {
  x[i] <- ifelse(rnorm(1) > 0, 1, 0)
  print(address(x))
}
```

```
## [1] "0x55cc53ed0b58"
## [1] "0x55cc53ed0c08"
## [1] "0x55cc53ed0cb8"
## [1] "0x55cc53ed0d68"
## [1] "0x55cc53ed0e18"
## [1] "0x55cc53ed0ec8"
## [1] "0x55cc40715180"
## [1] "0x55cc55478be0"
## [1] "0x55cc553dec60"
## [1] "0x55cc42901410"
```

Exercice 2.2

Comparer les temps de calcul des trois expressions suivantes. Représenter

graphiquement la variation des temps de calcul.

```
n <- 10 ^ 6
# expression 1
x <- numeric(n)
for (k in 1:n)
  x[k] <- (5 == sample(1:10, 1))
mean(x)
# expression 2
x <- NULL
for (k in 1:n)
  x <- c(x, (5 == sample(1:10, 1)))
mean(x)
# expression 3
x <- 0
for (k in 1:n)
  x <- x + (5 == sample(1:10, 1))
mean(x)
```

3.4 Fonction vectorisée

R est un langage interprété et de ce fait, l'exécution de boucles est coûteuse en temps calcul, contrairement à des langages compilés. Pour palier ce problème, les concepteurs du langage **R** ont créé de nombreuses fonctions vectorisées, c'est-à-dire qui peuvent s'appliquer à des vecteurs et qui font en général appel à des fonctions codées en **C** ou **Fortran** pour gagner en rapidité. Pour comprendre ce concept, on va comparer les deux méthodes suivantes dont le but est de calculer la somme des éléments d'un vecteur simulée selon une loi $\mathcal{N}(0, 1)$.

- Méthode 1 : on utilise des boucles depuis **R**

```
vec <- rnorm(1000000)
my_sum <- function(x) {
  res <- 0
  for (k in seq_along(vec)) {
    res <- res + vec[k]
  }
  return(res)
}
```

- Méthode 2 : on utilise la fonction `sum()` sur un vecteur. Cette dernière va bien entendu effectuer une boucle, mais elle sera effectuée en langage compilé, donc plus rapidement.

```
microbenchmark::microbenchmark(
  my_sum(vec),
  sum(vec),
```

```

times = 10L)

## Unit: microseconds
##          expr      min     lq    mean   median      uq     max neval
## my_sum(vec) 79171.798 79745.959 81540.340 80752.182 82759.752 86821.961    10
## sum(vec)    975.122   978.605 1048.408 1005.526 1078.631 1329.366    10

```

Remarque : il existe de nombreuses fonctions de base qui sont vectorisées et qui permettent de gagner en temps de calcul et on les utilise souvent sans s'en rendre compte. Par exemple, les fonctions *log()*, *cos()*, *exp()*, etc. Dès lors qu'on est amener à faire des boucles, cela peut valoir le coup de se poser la question : est-ce qu'il n'existe pas de fonctions prédéfinies et vectorisées qui feraient déjà ce qu'on souhaite faire, en un temps plus rapide.

Exercice 2.3

Ecrire la fonction *my_sd()* qui calcule l'écart-type d'un vecteur **numeric**. La fonction ne pourra pas faire appel ni à la fonction *sum()* ni *mean()* et ne pourra avoir qu'une seule boucle. Comparer les temps de calcul avec *sd()*.

3.5 Ecrire un code en C++

Lorsqu'on a identifié certaines parties d'un code qui sont coûteuses en temps calcul, il peut être intéressant de coder ces parties sous formes de fonctions écrites en **C++** et d'appeler ensuite ces fonctions depuis **R**.

Par exemple, on va convertir la fonction *my_sum()* écrite en **R** en code **C++**. On présente ci-dessous le code inclus dans le fichier *sumplusplus.cpp*.

```

#include <Rcpp.h> // pour utiliser des fonctions de base
using namespace Rcpp;

// La commande suivante permet d'appeler ensuite les fonctions
// créées depuis R

// [[Rcpp::export]]

// début de la fonction
double sum_rcpp(NumericVector x) {
  double res = 0;
  int n = x.size();

  for(int i = 0; i < n; i++) {
    res = res + x(i);
  }

  return res;
}

```

{}

L'algorithme utilisé dans la fonction `sum_rcpp()` présente la même syntaxe que la fonction `my_sum()` écrite en **R**. Par contre, il faut utiliser les particularités du langage **C++**. Parmi ces particularités :

- il faut définir le type de l'objet qui sera retourné par la fonction,
- de manière générale, il faut définir le type de tous les objets créés et/ou utilisés. En effet, on a du définir le type de tous les paramètres d'entrée. Même dans la boucle **for**, on a définir le type de **i** qui sert d'incrément,
- les lignes de code se terminent par le point virgule,
- l'opérateur d'affectation est le symbole `=`,
- la boucle **for** s'utilise différemment que du code **R**,
- pour connaître la taille d'un vecteur, on a utilisé la méthode `.size()` qui retourne un entier,
- l'indexation du vecteur commence à 0 et on utiliser les parenthèses pour accéder à un élément du vecteur.

Une fois la fonction écrite, il faut la charger sous **R**. Pour cela, on utilise la fonction `sourceCpp()` du package **Rcpp** (Eddelbuettel et al., 2019) qui est le package **R** le plus utilisé par d'autres packages.

```
require("Rcpp")
sourceCpp("R/sumcplusplus.cpp")
```

On peut ensuite appeler directement la fonction créée :

```
sum_rcpp(vec)
```

```
## [1] 270.4123
```

Depuis **R Markdown**, il est possible d'insérer du code écrit **C++** directement à l'intérieur d'un chunk. Pour cela, il suffit de cliquer depuis **RStudio** sur l'onglet “Insert”, puis “Rcpp” et cela aura pour effet d'insérer une balise de code qui sera compilée au moment de son évaluation (autrement dit, `sourceCpp()` est automatiquement exécutée).

Si on compare les temps de calcul avec `my_func()` et la fonction `sum()`, on constate que `sum_rcpp()` est presque 20 fois plus rapides que `my_sum()` et 4 fois plus lentes que `sum()`.

```
microbenchmark::microbenchmark(
  my_sum(vec),
  sum(vec),
  sum_rcpp(vec),
  times = 10L)
```

3.6. INSÉRER DU CODE PYTHON DANS UN DOCUMENT MARKDOWN71

```
## Unit: microseconds
##           expr      min       lq     mean    median      uq     max
##   my_sum(vec) 78677.186 78849.071 79794.805 79642.6285 80081.235 82813.024
##   sum(vec)    974.451   975.041  1094.846   976.0095  985.024  1974.992
## sum_rcpp(vec) 3888.509  3889.551  4192.511  3982.4395  4051.475  6117.131
## neval
##      10
##      10
##      10
```

Pour en savoir plus sur l'utilisation de code **C++** depuis **R**, on recommande le document suivant : <http://adv-r.had.co.nz/Rcpp.html>

Exercice 2.4

Ecrire la fonction *my_sd_cpp()* en **C++** qui calcule l'écart-type d'un vecteur **numeric**. Comparer les temps de calcul avec *sd()*.

Remarque : en **C++**, on utilise *pow(a, b)* pour calculer a^b

3.6 Insérer du code Python dans un document Markdown

Il est possible d'exécuter du code **Python** dans un document **R** Markdown et d'importer ensuite les objets créés sous **R** pour les utiliser.

Pour cela, il suffit de cliquer depuis **RStudio** sur l'onglet Insert, puis Python et cela aura pour effet d'insérer une balise de code Python

```
```{python}
````
```

Par exemple, on va executer les lignes de commande **Python** suivantes dans lesquelles on a créé l'objet **flights** :

```
import pandas
flights = pandas.read_csv("http://www.thibault.laurent.free.fr/cours/R_avance/flights.csv")
flights = flights[flights['Dest'] == "TPA"]
flights = flights[['UniqueCarrier', 'DepDelay', 'ArrDelay']]
flights = flights.dropna()
```

Pour rappatrier l'objet **Python** dans **R**, il suffit ensuite d'utiliser la commande **py\$** du package **reticulate** (Ushey et al., 2019) suivi de l'objet à rappatrier :

```
library("reticulate")
library("ggplot2")
ggplot(py$flights, aes(UniqueCarrier, ArrDelay)) +
  geom_point() +
  geom_jitter()
```

3.7 Eviter si possible les boucles (*apply()*, *lapply()*, *replicate()*, *colSums()*, etc.)

On va parler ici des fonctions *apply()*, *lapply*, *sapply()*, *mapply()*, *tapply()*. Les différences en temps de calcul ne sont pas significatives par rapport à l'utilisation de boucles, dans la mesure où implicitement, ces fonctions font le même travail que les boucles. L'idée de ces fonctions est de réduire le nombre de lignes de code qu'on utiliserait en faisant des boucles à la place.

On va voir qu'on utilise une de ces fonctions plutôt qu'une autre selon le type de l'objet considéré en entrée et selon le type d'objets que l'on souhaite en sortie. On verra également la fonction *replicate()* qui utilise les propriétés de la fonction *sapply()*, mais pour un usage un peu différent.

On présentera également les fonctions *colSums()*, *rowSums()*, *colMeans()*, *rowMeans()* qui sont plus performantes qu'utiliser la fonction *apply()* car elles sont vectorisées.

3.7.1 *apply()* pour les matrix/array

Une **matrix** est une forme particulière d'un **array**. Une **matrix** possède deux dimensions (espace des lignes et espace des colonnes), un **array** peut avoir autant de dimensions que souhaitées. Les D -dimensions sont données sous forme d'un vecteur de taille D , chaque élément i donnant la taille de la dimension i . En général, on utilise peu les **array** à plus de 3 dimensions. Une caractéristique d'un **array** est que tous ses éléments sont du même type : **numeric**, **integer**, **logical**, **character** (rarement **complex**).

Commençons par créer un **array** à trois dimensions de taille $3 \times 2 \times 2$. Pour simplifier la compréhension d'un **array**, on va donner des noms à chaque composante de chaque dimension. La 1ère dimension correspondra aux nom des individus, la 2ème celle des variables, la 3ème celle des années :

```
tab <- array(c(100, 90, 110, 25, 24, 28, 175, 180, 190, 68, 74, 85),
              dim = c(3, 2, 2))
dimnames(tab)[[1]] <- c("Luc", "Pierre", "Pedro")
dimnames(tab)[[2]] <- c("taille", "poids")
dimnames(tab)[[3]] <- c("2000", "2010")
tab

## , , 2000
##
##      taille poids
## Luc      100    25
## Pierre   90     24
## Pedro    110    28
##
## , , 2010
```

```
##           taille poids
## Luc        175   68
## Pierre     180   74
## Pedro      190   85
```

La fonction *apply()* permet de faire des calculs :

- sur une seule dimension. On précise alors le numéro de la dimension dans le deuxième argument et le troisième argument correspond à la fonction qu'on souhaite appliquer sur chaque élément appartenant à cette dimension. Dans ce cas, le résultat est un vecteur de longueur la taille de la dimension choisie. Par exemple, si on souhaite faire la moyenne des éléments en considérant la 2ème dimension.

```
apply(tab, 2, mean)

##    taille    poids
## 140.83333 50.66667
```

Remarque : en réalité, selon la fonction qu'on applique à *apply()*, le résultat n'est pas forcément un vecteur. Essayer par exemple de remplacer la fonction *mean()* par *range()* dans l'exemple ci-dessus.

Equivalent avec les boucles : si on avait utilisé les boucles plutôt que la fonction *apply()*, on aurait utilisé le code suivant :

```
res <- numeric(dim(tab)[2])
for (k in seq_along(res))
  res[k] <- mean(tab[, k, ])
res
```

```
## [1] 140.83333 50.66667
```

- sur deux dimensions :

```
apply(tab, c(2, 3), mean)
```

```
##           2000      2010
## taille 100.00000 181.66667
## poids   25.66667  75.66667
```

Equivalent avec les boucles : si on avait utilisé les boucles pour faire ce calcul, on aurait fait :

```
res <- array(0, dim = dim(tab)[c(2, 3)])
for (i in 1:dim(tab)[2])
  for (j in 1:dim(tab)[3])
    res[i, j] <- mean(tab[, i, j])
res
```

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

```
## [1,] 100.00000 181.66667
## [2,] 25.66667 75.66667
```

3.7.2 Les fonctions `colSums()`, `rowSums()`, `colMeans()`, `rowMeans()`

Ces fonctions sont équivalentes à la fonction `apply()` en utilisant **FUN=sum** ou **FUN=mean** et en appliquant les bonnes dimensions (1 pour lignes, 2 pour colonnes), mais sont plus rapides car elles ont été codées en langage compilé.

Pour comparer les temps de calcul :

```
x <- matrix(runif(10e6), nc = 5)
microbenchmark::microbenchmark(
  apply(x, 2, mean),
  colMeans(x),
  times = 10L)

## Unit: milliseconds
##          expr      min       lq     mean   median      uq     max
##  apply(x, 2, mean) 183.02811 196.58432 232.10561 196.79185 197.63764 546.86843
##  colMeans(x)    11.29878 11.37811 11.47504 11.45622 11.50457 11.78339
##  neval
##      10
##  times
```

3.7.3 La fonction `lapply()`

Même si la fonction `lapply()` s'applique aussi bien sur les vecteurs que les **list**, on présentera ici son utilisation pour les **list**/**data.frame**. L'intérêt de l'appliquer sur des vecteurs sera présenté avec la fonction `replicate()`.

On rappelle ici qu'une **list** est caractérisée par un certain nombre d'éléments pouvant être de types différents. On accède aux éléments d'une liste par le symbole `$` suivi du nom de l'élément de la liste et s'il n'y a pas de nom, on y accède avec les doubles crochets avec l'indice de l'élément auquel on souhaite accéder. Un **data.frame** est une forme particulière d'une **list** dans la mesure où les éléments peuvent être vus comme étant les variables (on y accède avec le symbole `$`).

La fonction `lapply()` consiste à appliquer la même opération sur chaque élément de la **list**. Cette opération pouvant être donnée par une fonction de base de **R** ou bien pouvant être une fonction programmée.

On considère le jeu de données **mtcars** accessible par défaut sous **R** et contenant un certaine nombre de variables sur des moteurs de voitures de différentes marques. Pour connaître les caractéristiques moyennes, on applique la fonction `mean()` à chaque élément de **mtcars**, autrement dit à chaque variable. Le résultat est retourné sous forme de **list** de même longeur que la **list** de départ.

3.7. EVITER SI POSSIBLE LES BOUCLES (APPLY(), LAPPLY(), REPLICATE(), COLSUMS(), ETC.)75

Ici, on a utilisé la fonction *unlist()* sur le résultat retourné afin de présenter le résultat sous forme d'un vecteur, plus lisible à lire :

```
unlist(lapply(mtcars, mean))

##      mpg      cyl      disp      hp      drat      wt      qsec
## 20.090625  6.187500 230.721875 146.687500  3.596563  3.217250 17.848750
##      vs      am      gear      carb
## 0.437500  0.406250  3.687500  2.812500
```

Cette fonction nous permet ainsi d'éviter de faire la boucle suivante :

```
res <- numeric(length(mtcars))
for (k in seq_along(res))
  res[k] <- mean(mtcars[[k]])
res

## [1] 20.090625  6.187500 230.721875 146.687500  3.596563  3.217250
## [7] 17.848750  0.437500  0.406250  3.687500  2.812500
```

3.7.4 La fonction *sapply()*

On a vu que la fonction *lapply()* retournait une **list**. Or, si on sait par avance que le résultat retourné pour chaque élément de la liste sera identique pour chaque élément, la fonction *sapply()* va concaténer les résultats de chaque élément sous forme d'un vecteur ou d'un tableau. Si on reprend l'exemple précédent :

```
sapply(mtcars, mean)

##      mpg      cyl      disp      hp      drat      wt      qsec
## 20.090625  6.187500 230.721875 146.687500  3.596563  3.217250 17.848750
##      vs      am      gear      carb
## 0.437500  0.406250  3.687500  2.812500
```

On présente ici un autre exemple où on va créer notre propre fonction qui consiste à retourner, le minimum, le maximum, la moyenne, l'écart-type et la moyenne de chaque élément. On code la fonction suivante où l'argument **x** sera interprété comme représentant un élément de la **list**.

```
f <- function(x)
  c(min = min(x), max = max(x), mean = mean(x), med = median(x), sd = sd(x))
```

On applique ensuite la fonction *f()* à chaque élément de la **list** avec la fonction *sapply()*. On transpose le résultat car c'est en général ainsi que sont présentées les statistiques descriptives des variables :

```
t(sapply(mtcars, f))
```

```
##      min      max      mean      med      sd
## mpg 10.400  33.900 20.090625 19.200  6.0269481
## cyl  4.000   8.000  6.187500  6.000  1.7859216
```

```
## disp 71.100 472.000 230.721875 196.300 123.9386938
## hp    52.000 335.000 146.687500 123.000  68.5628685
## drat   2.760   4.930   3.596563   3.695   0.5346787
## wt     1.513   5.424   3.217250   3.325   0.9784574
## qsec   14.500  22.900  17.848750  17.710   1.7869432
## vs      0.000   1.000   0.437500   0.000   0.5040161
## am      0.000   1.000   0.406250   0.000   0.4989909
## gear    3.000   5.000   3.687500   4.000   0.7378041
## carb    1.000   8.000   2.812500   2.000   1.6152000
```

Par ailleurs, pour insérer une table dans un document Markdown, on peut utiliser la fonction `tbl()` du package **kableExtra** dont on verra d'autres exemples dans le chapitre sur la visualisation de données.

```
kableExtra::tbl(t(round(sapply(mtcars, f), 3)))
```

| | min | max | mean | med | sd |
|------|--------|---------|---------|---------|---------|
| mpg | 10.400 | 33.900 | 20.091 | 19.200 | 6.027 |
| cyl | 4.000 | 8.000 | 6.188 | 6.000 | 1.786 |
| disp | 71.100 | 472.000 | 230.722 | 196.300 | 123.939 |
| hp | 52.000 | 335.000 | 146.688 | 123.000 | 68.563 |
| drat | 2.760 | 4.930 | 3.597 | 3.695 | 0.535 |
| wt | 1.513 | 5.424 | 3.217 | 3.325 | 0.978 |
| qsec | 14.500 | 22.900 | 17.849 | 17.710 | 1.787 |
| vs | 0.000 | 1.000 | 0.438 | 0.000 | 0.504 |
| am | 0.000 | 1.000 | 0.406 | 0.000 | 0.499 |
| gear | 3.000 | 5.000 | 3.688 | 4.000 | 0.738 |
| carb | 1.000 | 8.000 | 2.812 | 2.000 | 1.615 |

Remarque : si on souhaite appliquer la fonction `f()` pour les éléments qui sont de type **numeric** et la fonction `table()` pour les éléments de type **factor**, alors la fonction `sapply()` retournera le même résultat que la fonction `lapply()`, à savoir une **list**, car les résultats retournés pour chaque élément ne seront pas les mêmes :

```
sapply(iris, function(x){
  if (is.numeric(x))
    f(x)
  else
    table(x)
})

## $Sepal.Length
##       min      max      mean      med      sd
## 4.3000000 7.9000000 5.8433333 5.8000000 0.8280661
##
## $Sepal.Width
```

```

##      min      max      mean      med      sd
## 2.0000000 4.4000000 3.0573333 3.0000000 0.4358663
##
## $Petal.Length
##      min      max      mean      med      sd
## 1.0000000 6.9000000 3.7580000 4.3500000 1.765298
##
## $Petal.Width
##      min      max      mean      med      sd
## 0.1000000 2.5000000 1.1993333 1.3000000 0.7622377
##
## $Species
## x
##   setosa versicolor  virginica
##      50          50          50

```

3.7.5 La fonction *replicate()*

On va partir de l'exemple suivant : on souhaite simuler 5 échantillons de taille 10 distribués selon une loi uniforme $U_{[0,1]}$.

- La première solution serait de faire une boucle :

```

res <- vector("list", 5)
for (k in 1:5)
  res[[k]] <- runif(10)

```

- La deuxième solution consiste à appliquer la fonction *lapply()* sur un vecteur quelconque de taille 5 (ici, le plus simple est de faire *1:5*, mais on aurait pu prendre n'importe quel vecteur de taille 5) et de lui appliquer une fonction qui retourne un vecteur simulé selon une $U_{[0,1]}$. Un vecteur est donc considéré ici comme une liste où chaque élément du vecteur serait un élément d'une liste.

```
res <- lapply(1:5, function(x) runif(10))
```

Remarque : dans le deuxième argument, la fonction qu'on va appliquer sur chaque élément d'un vecteur prend comme argument d'entrée **x** (un élément du vecteur), mais on ne l'utilise pas à l'intérieur de la fonction, car on n'en a pas besoin pour faire ce qu'on souhaite faire.

- La troisième solution consiste à utiliser la fonction *replicate()*, qui comme la fonction *sapply()* va retourner le résultat sous une forme plus simplifiée qu'une **list** :

```
res <- replicate(5, runif(10))
```

Remarque : la fonction *replicate()* fait précisément appel à la fonction *sapply()*. Pour cela, elle crée un vecteur de 0L de taille le premier argument de *replicate()*.

3.7.6 La fonction *mapply()*

Pour illustrer cette fonction, on considère 5 échantillons de taille 10 issus d'une loi $U_{[0,1]}$. Pour calculer la moyenne par échantillon, on peut utiliser la fonction *lapply()* :

```
xs <- replicate(5, runif(10), simplify = FALSE)
lapply(xs, mean)
```

```
## [[1]]
## [1] 0.4200745
##
## [[2]]
## [1] 0.4105398
##
## [[3]]
## [1] 0.363235
##
## [[4]]
## [1] 0.5592183
##
## [[5]]
## [1] 0.4279419
```

A présent, supposons qu'on souhaite pondérer les moyennes par un vecteur de poids qui est différent selon chaque échantillon. On crée ici les poids associés à chaque échantillon sous forme d'une liste :

```
ws <- replicate(5, rpois(10, 5) + 1, simplify = FALSE)
```

Une façon de faire avec des boucles serait la suivante :

```
res <- numeric(length(xs))
for (k in seq_along(res))
  res[k] <- sum(xs[[k]] * ws[[k]]) / sum(ws[[k]])
res

## [1] 0.3918269 0.4219046 0.3278862 0.5367989 0.4524113
```

Le problème de la fonction *lapply()* est qu'on ne peut pas appliquer une fonction sur deux **list** simultanément. C'est ce que fait la fonction *mapply()* qui prend comme 1er argument d'entrée une fonction à appliquer sur les éléments d'autant de **list** que l'on souhaite. La fonction qu'on applique contient donc deux arguments d'entrée **x** et **y** qui correspondent aux éléments de chacune des deux listes :

```
mapply(function(x, y) sum(x * y) / mean(y), xs, ws)
```

```
## [1] 3.918269 4.219046 3.278862 5.367989 4.524113
```

3.7.7 La fonction *tapply()*

On considère les deux vecteurs suivants :

```
(taille <- c(rnorm(5, 165, 10), rnorm(5, 175, 10)))
## [1] 141.7006 181.0656 164.2458 177.7004 174.7507 165.0026 180.6391 179.8185
## [9] 170.9057 177.5540
(sexe <- rep(c("M", "F"), each = 5))
## [1] "M" "M" "M" "M" "M" "F" "F" "F" "F" "F"
```

On souhaite calculer la moyenne dans chaque sous-groupe “M” et “F”. Une façon de faire est de créer une **list** contenant deux éléments, le premier correspondant au vecteur de **taille** du sous-groupe “M” et le second correspondant au vecteur de **taille** du sous-groupe “F”. C'est ce que fait la fonction *split()* ci-dessous :

```
(res.split <- split(taille, sexe))
## $F
## [1] 165.0026 180.6391 179.8185 170.9057 177.5540
##
## $M
## [1] 141.7006 181.0656 164.2458 177.7004 174.7507
```

Ensuite, il suffit d'appliquer la fonction *sapply()* à cette **list** :

```
sapply(res.split, mean)
##          F          M
## 174.7840 167.8926
```

La fonction *tapply()* permet de faire ce calcul en une seule ligne :

```
tapply(taille, sexe, mean)
##          F          M
## 174.7840 167.8926
```

Remarque : la fonction *tapply()* ne s'appliquant que sur des vecteurs, la fonction *by()* permet de généraliser la fonction *tapply()* aux **data.frame**.

Exercice 2.5

- Créer deux fonctions *mean_rnorm.1()* et *mean_rnorm.2()* qui prennent en entrée les paramètres **n** et **p**. Ces fonctions permettent de simuler **n** échantillons de taille **p** de lois normales centrées réduites et retournent pour chaque échantillon la moyenne. *mean_rnorm.1()* utilisera des boucles, *mean_rnorm.2()* la fonction *replicate()*.
- Comparer les temps de calcul de ces 2 fonctions.

3.8 Améliorer ses fonctions

3.8.1 Créer des sous-fonctions

Il ne faut pas hésiter à créer des petites fonctions qui peuvent être

- locales si elles ne sont utilisées qu'à l'intérieur d'une seule fonction principale
- globales si elles ont à vocation d'être utilisées dans plusieurs fonctions.

Supposons par exemple qu'on souhaite calculer une densité non paramétrique de la densité en utilisant un des noyaux suivants :

- noyau *biweight* $K(x) = \frac{15}{16}(1 - (\frac{x}{h})^2)^2 1_{(\frac{x}{h})^2 \leq 1}$
- noyau *triweight* $K(x) = \frac{35}{32}(1 - (\frac{x}{h})^2)^3 1_{(\frac{x}{h})^2 \leq 1}$
- noyau gaussien $K(x) = \frac{1}{\sqrt{2\pi}} \exp(-0.5(\frac{x}{h})^2)$

où **h** est le paramètre de lissage. Dans un premier temps, on peut alors programmer une fonction par noyau :

```
biweight <- function(x, h) {
  return(15 / 16 * (1 - (x / h) ^ 2) ^ 2 * ifelse((x / h) ^ 2 <= 1, 1, 0))
}
triweight <- function(x, h) {
  return(35 / 32 * (1 - (x / h) ^ 2) ^ 3 * ifelse((x / h) ^ 2 <= 1, 1, 0))
}
gaussian <- function(x, h) {
  return(1 / sqrt(2 * pi) * exp(-0.5 * (x / h) ^ 2))
```

3.8.2 Structure de contrôle

Les structures de contrôle classiques (**for**, **while**, **repeat**, **if/else**) sont bien entendu disponibles dans **R** (voir les documents “Pour se donner un peu d’**R**” et “Introduction à **R**”). Nous nous intéressons ici à la notion d’aiguillage (fonction **switch()**) moins couramment utilisée bien que très pratique.

Celle-ci permet d’éviter d’emboîter des **if/else** lorsqu’on a plusieurs options possibles. Supposons par exemple qu’on souhaite calculer une densité non paramétrique de la densité en utilisant un des noyaux suivants :

- noyau *biweight* $K(x) = \frac{15}{16}(1 - (\frac{x}{h})^2)^2 1_{(\frac{x}{h})^2 \leq 1}$
- noyau *triweight* $K(x) = \frac{35}{32}(1 - (\frac{x}{h})^2)^3 1_{(\frac{x}{h})^2 \leq 1}$
- noyau gaussien $K(x) = \frac{1}{\sqrt{2\pi}} \exp(-0.5(\frac{x}{h})^2)$

où **h** est le paramètre de lissage. On peut alors programmer sous **R** la fonction *f_noyau()* suivante, prenant en entrée les arguments **x**, **h** et **type** où **type** est un caractère prenant les valeurs “**bi**”, “**tri**” et “**gauss**”.

3.8.2.1 Solution avec if/else

La façon traditionnelle pour gérer différents cas de figure est d'utiliser les conditions **if/else** :

```
f_noyau <- function(x, h, type = "bi") {
  if (type == "bi") {
    biweight(x, h)
  } else {
    if (type == "tri") {
      triweight(x, h)
    } else {
      gaussian(x, h)
    }
  }
}
```

Un inconvénient est qu'en emboîtant des conditions **if/else**, on peut alors vite se perdre dans la lecture du code.

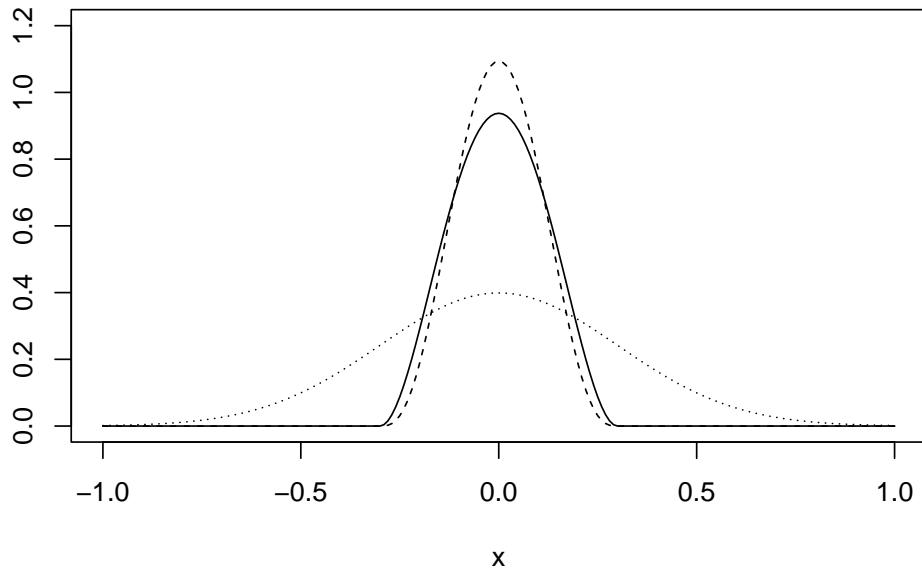
3.8.2.2 Solution avec switch()

La fonction *switch()* s'applique sur l'argument **type** et on donne ensuite pour chaque valeur possible de **type** le type de calcul à effectuer. On ajoute ici un cas qui correspondrait à toutes autres valeurs de **type**.

```
f_noyau.2 <- function(x, h, type = "bi") {
  switch(type, bi = biweight(x, h),
         tri = triweight(x, h),
         gauss = gaussian(x, h),
         "Préciser une autre valeur de type")
}
```

Application :

```
x <- seq(-1, 1, 0.01)
plot(x, f_noyau.2(x, 0.3, type = "bi"), type = "l", ylab = "", ylim = c(0, 1.2))
lines(x, f_noyau.2(x, 0.3, type = "tri"), lty = 2)
lines(x, f_noyau.2(x, 0.3, type = "gauss"), lty = 3)
```



```
f_noiayu.2(x, 0.3, type = "unif")
## [1] "Préciser une autre valeur de type"
```

3.8.3 Fonction *stopifnot()*

On connaît la fonction *stop()* qui s'insère en général à l'intérieur des conditions *if/else*.

La fonction *stopifnot()* permet de tester simultanément plusieurs conditions et évite donc d'avoir recours aux *if/else*. Elle est particulièrement utile à l'intérieur d'une fonction pour, par exemple, vérifier la conformité des valeurs passées en paramètres. Un exemple d'utilisation :

```
stopifnot(1 < 2, length(1:2) == 2, pi < 2, cos(pi) > 3)
```

```
## Error: pi < 2 n'est pas TRUE
```

Si plusieurs conditions ne sont pas respectées, c'est la première non respectée qui est renvoyée comme source de l'erreur.

Exercice 2.6

Modifier la fonction *f_noiayu.2()* définie précédemment pour vérifier que :

- le paramètre **x** est **numeric**
- **h** est de taille 1 et de type **numeric**
- **type** est une chaîne de caractères (on supprimera la possibilité de passer un entier pour choisir le type).

3.8.4 Les arguments dans une fonction

3.8.4.1 Supprimer les paramètres inutiles

Dans l'exemple ci-dessous, les paramètres `d` et `e` ne sont pas utilisés dans la fonction et pourtant ils ont un coût de stockage et un coût en temps de calcul car ils sont évalués par défaut dans la fonction.

```
f_1 <- function(a = 5, b = 4, d = 3, e = 1)
  (a + b)^2
```

On compare les temps de calcul avec la même fonction mais qui n'utilise pas les paramètres inutilisés :

```
f_2 <- function(a = 5, b = 4)
  (a + b)^2

microbenchmark::microbenchmark(
  f_1(),
  f_2()
)

## Unit: nanoseconds
##   expr  min    lq      mean    median     uq      max   neval
##   f_1() 480 525.5 27955.93    587 772.0 2706845    100
##   f_2() 362 415.5 32974.04    479 635.5 3227319    100
```

Contrairement à d'autres langages, R ne signale pas les paramètres qui ne sont pas utiles, qu'il s'agisse de paramètres utilisés en argument d'entrée ou de paramètres créés à l'intérieur de la fonction. C'est donc à celui qui programme d'être vigilant à ne pas créer des paramètres inutiles.

3.8.4.2 Utiliser des fonctions comme argument d'entrée

On a vu dans la section précédente que la fonction `lapply()` avait dans ses argument d'entrée une fonction. On peut si on le souhaite créer nos propres fonctions qui ont comme argument d'entrée une fonction. Dans l'exemple suivant, on applique une fonction choisie par l'utilisateur sur un échantillon simulée selon une loi uniforme :

```
randomise <- function(f) f(runif(1e3))
```

Pour l'utiliser, on remplace l'argument `f` par le nom de la fonction qui nous intéresse :

```
randomise(mean)

## [1] 0.4924948
randomise(mean)

## [1] 0.5048686
```

```
randomise(sum)
```

```
## [1] 496.2882
```

3.8.4.3 Utiliser des fonctions comme argument de sortie

Il est également possible de faire retourner une fonction par une fonction. C'est ce que fait la fonction *f_power()* ci-dessous :

```
f_power <- function(exponent)
  function(x) x^exponent
```

Pour utiliser cette fonction :

```
f_power(2)(1:5)
```

```
## [1] 1 4 9 16 25
```

```
f_power(3)(1:5)
```

```
## [1] 1 8 27 64 125
```

On peut également créer les fonctions *f_square()* et *f_cube()* à partir de la fonction *f_power()* :

```
f_square <- f_power(2)
f_cube <- f_power(3)
class(f_square)
```

```
## [1] "function"
```

```
class(f_cube)
```

```
## [1] "function"
```

Et pour obtenir le calcul souhaité :

```
f_square(1:5)
```

```
## [1] 1 4 9 16 25
```

```
f_cube(1:5)
```

```
## [1] 1 8 27 64 125
```

3.8.4.4 Utiliser les ... comme argument d'entrée

Lorsqu'un utilisateur définit une fonction, il peut permettre à sa fonction d'utiliser toutes les options d'une autre fonction sans les lister une à une. Prenons l'exemple de la fonction *plot_reg()* définie ci-dessous.

```
plot_reg <- function(x, y, np = TRUE, ...) {
  plot(y ~ x, ...)
```

```

abline(lm(y ~ x), col = "blue")
if (np) {
  np.reg <- loess(y ~ x)
  x.seq <- seq(min(x), max(x), length.out = 25)
  lines(x.seq, predict(np.reg, x.seq), col = "red")
}
}
```

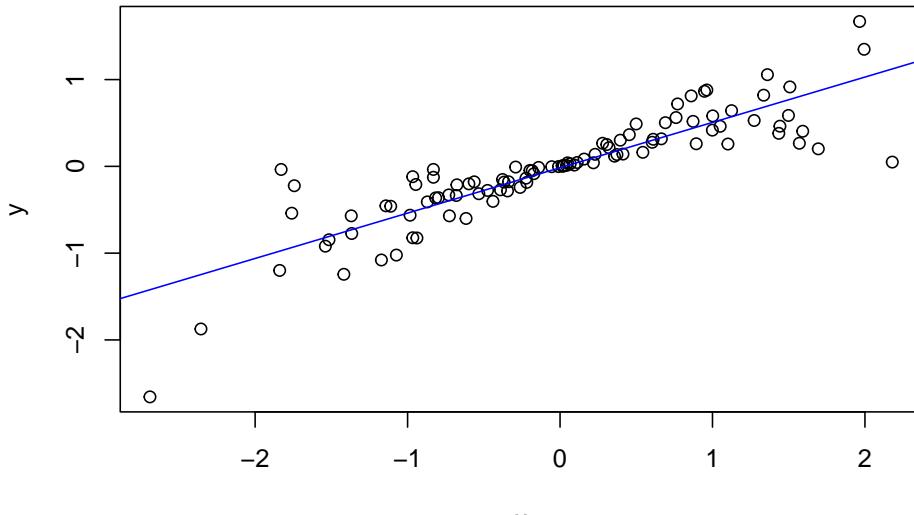
L'utilisation de la syntaxe ... permet de préciser que la fonction *plot_reg()* pourra, si besoin, faire appel à n'importe quelle option de la fonction *plot()*.

Pour illustrer la chose, exécuter les instructions suivantes.

```

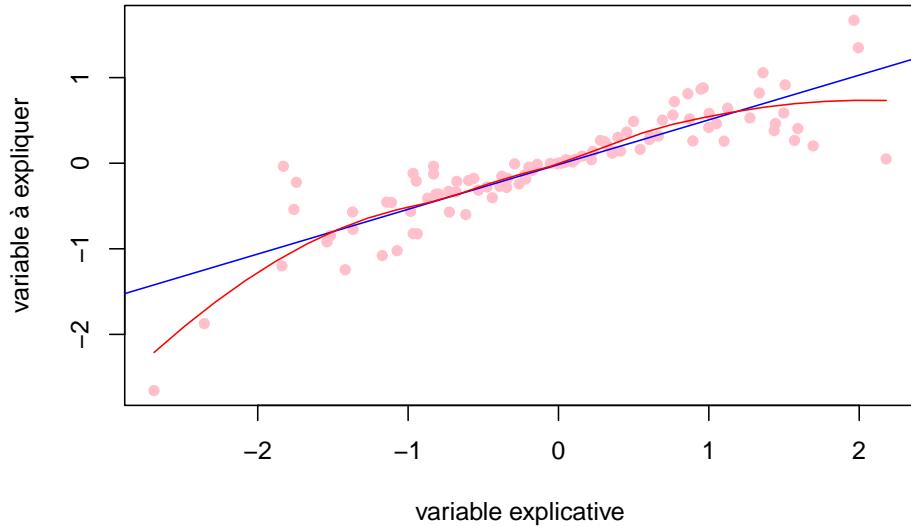
x <- rnorm(100)
y <- x*runif(100)

plot_reg(x, y, np = FALSE)
```



```

plot_reg(x, y, pch = 16, col = "pink",
         xlab = "variable explicative", ylab = "variable à expliquer")
```



Exercice 2.7

Q1 Que fait la fonction *plot_reg()* ?

Q2 Ecrire une fonction *hist_extrem()* qui prend en argument un entier **n**, un entier **B** et ... qui correspondra aux options utilisées dans la fonction *hist()*. Cette fonction devra mettre en oeuvre l'algorithme ci-dessous :

Répéter **B** fois l'opération suivante :

- simulation d'un vecteur **x** de taille **n** selon une $N(0, 1)$,

Résultat On comptabilise sur les **B** boucles le pourcentages de boucles où il existe au moins un élément de **x** supérieur en valeur absolue à 1.96. On stockera les valeurs concernées et on représentera l'histogramme des valeurs extrêmes.

3.9 A quoi servent les fonctions *substitute()*/*quote()* et *eval()* ?

On considère l'exemple ci-dessous :

```
a <- 5
```

```
identical(a, "a")
```

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

a et “**a**” sont bien entendu deux entités différentes. **a** est un objet qui pointe sur la valeur 5 alors que “**a**” est une chaîne de caractère.

Losqu'on charge une librairie, on a la possibilité d'utiliser les deux commandes suivantes :

```
require("tidyverse")
require(tidyverse)
```

Dans le deuxième cas, **tidyverse** (sans guillemet) n'est a priori pas un objet (à moins qu'au cours d'une session on ait créé un objet avec ce nom). C'est donc qu'il a été interprété comme étant quelque chose de différent d'un objet.

3.9.1 Les fonctions *quote()* et *substitute()*

Les fonctions *quote()* et *substitute()* permettent de capturer une instruction sans l'évaluer. Pour généraliser, on va appeler cette capture une expression. La différence entre *eval()* et *substitute()* est décrite dans le chapitre Non-standard evaluation de l'ouvrage d'Hadley Wickham, mais pour résumer *quote()* ne fait que retourner l'expression qu'on lui donne en entrée alors que *substitute* est un peu plus complexe.

```
quote(1:10)

## 1:10
quote(f(1:10))

## f(1:10)
quote(f(x))

## f(x)
quote(f(x + y^2))

## f(x + y^2)
substitute(1:10)

## 1:10
substitute(f(1:10))

## f(1:10)
substitute(f(x))

## f(x)
substitute(f(x + y^2))

## f(x + y^2)
```

Une fois qu'on a fait cette capture de code, le but est d'évaluer cette expression dans un environnement spécifique et au moment où on le souhaitera avec la fonction *eval()*.

3.9.2 La fonction `eval()`

Cette fonction permet d'évaluer une expression. Par défaut, elle est évaluée dans l'environnement courant de **R**. Autrement dit, si on considère l'expression suivante :

```
a <- quote(f.eval(x.eval))
```

Si on veut évaluer cette expression, il faut que la fonction `f.eval()` et l'objet `x.eval` existent tous les deux dans l'environnement où on souhaite exécuter cette expression. Par défaut, `eval()` ira chercher dans l'environnement courant. Ici, on obtient le message d'erreur suivant car la fonction `f.eval()` et l'objet `x.eval` n'ont pas été définies au préalables :

```
eval(a)
```

```
## Error in f.eval(x.eval): impossible de trouver la fonction "f.eval"
```

On considère un autre exemple avec l'expression suivante :

```
a <- quote(Species == "setosa")
```

La commande suivante donnera un message d'erreur :

```
eval(a)
```

```
## Error in eval(a): objet 'Species' introuvable
```

En revanche, on peut préciser qu'on va trouver le nom `Species` dans l'environnement du `data.frame` nommé `iris` :

```
eval(a, envir = iris)
```

```
## [1] TRUE TRUE
## [13] TRUE TRUE
## [25] TRUE TRUE
## [37] TRUE TRUE
## [49] TRUE TRUE FALSE FALSE
## [61] FALSE FALSE
## [73] FALSE FALSE
## [85] FALSE FALSE
## [97] FALSE FALSE
## [109] FALSE FALSE
## [121] FALSE FALSE
## [133] FALSE FALSE
## [145] FALSE FALSE
```

3.9.2.1 Et en pratique ?

Elles sont utilisées dans beaucoup de fonction sans qu'on s'en apperçoive nécessairement. On va voir quelques fonctions qui utilisent ces expressions :

- dans la fonction `subset()`, le deuxième argument est considérée comme une expression :

```
subset(iris, Species == "setosa" & Sepal.Length > 5.5)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 15      5.8      4.0       1.2       0.2  setosa
## 16      5.7      4.4       1.5       0.4  setosa
## 19      5.7      3.8       1.7       0.3  setosa
```

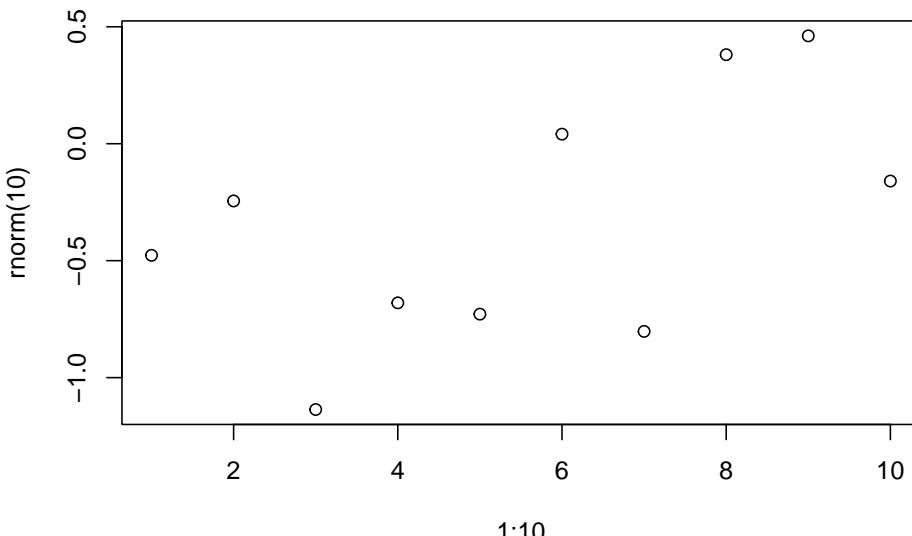
Ainsi, l'expression `Species == "setosa"` & `Sepal.Length > 5.5` sera évaluée dans le jeu de données `iris` ce qui permet d'éviter de faire :

```
iris[iris$Species == "setosa" & iris$Sepal.Length > 5.5, ]
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 15      5.8      4.0       1.2       0.2  setosa
## 16      5.7      4.4       1.5       0.4  setosa
## 19      5.7      3.8       1.7       0.3  setosa
```

- les différentes fonctions `filter()`, `arrange()`, etc. du package `dplyr` présentées dans le chapitre précédent utilisent le même genre de syntaxe que celui de la fonction `subset()`.
- la fonction `plot.default()` qui représente un nuage de points :

```
plot(1:10, rnorm(10))
```



Les arguments par défaut pour représenter la légende en abscisse et en ordonnées (`xlab` et `ylab`) correspondent aux expressions des paramètres `x` et `y` données en entrée.

3.9.3 Débugger une fonction

3.9.3.1 Analyser le message d'erreur

Lorsqu'un message d'erreur (ou d'avertissement) apparaît en rouge dans la console, la première chose à faire est d'essayer de le comprendre. La plupart du temps, le message est suffisamment implicite pour qu'on puisse trouver d'où provient le problème.

```
sum(c("a", "b"))
```

```
## Error in sum(c("a", "b")): 'type' (character) de l'argument incorrect
```

Dans cet exemple, le message d'erreur n'intervient pas dans le cœur de la fonction, mais dans l'appel des arguments. Lorsque l'erreur est dûe à l'exécution d'une commande à l'intérieur d'une fonction, il est parfois plus difficile d'identifier d'où vient le problème.

3.9.3.2 La fonction *traceback()*

Le principe de la fonction *traceback()* est le suivant : si un message d'erreur est apparu à cause de l'appel d'une commande, l'historique qui a conduit à l'appel de cette commande sera affiché.

Ici, on crée une fonction *ex_bug()* à partir des fonctions *mean_rnorm.1()* et *mean_rnorm.2()* créées précédemment, dans laquelle on a glissé une erreur.

```
ex_bug <- function(n, p) {
  n2 <- as.character(n)
  length(which(mean_rnorm.1(n2, p) > mean_rnorm.2(n, p)))
}
```

On obtient le message d'erreur suivant :

```
ex_bug(5, 10)
```

```
## Error in mean_rnorm.1(n2, p): impossible de trouver la fonction "mean_rnorm.1"
```

On appelle la fonction *traceback()* dans la foulée du message d'erreur :

```
traceback()

4: matrix(0, n, 1) at #2
3: mean_rnorm.1(n2, p) at #3
2: which(mean_rnorm.1(n2, p) > mean_rnorm.2(n, p)) at #3
1: ex_bug(5, 10)
```

Comment interpréter le résultat de la fonction *traceback()*. Le message d'erreur est intervenu au moment d'appeler la commande de la ligne 4. Cette commande a été appelée à la suite de l'appel de la commande de la ligne 3. La commande de la ligne 3 a été appelée suite à la commande de la ligne 2. La ligne 1 correspond bien entendu à l'appel de la fonction de départ *ex_bug()*.

3.9.3.3 La fonction `debugonce()`

La fonction `debugonce()` a pour objectif de rentrer dans le cœur de la fonction considérée et d'exécuter les lignes de code l'une après l'autre. Après chaque exécution, il est possible d'utiliser la fonction `ls()`, `print()`, etc. pour connaître le résultat des étapes intermédiaires. On n'est pas obligé d'utiliser cette fonction uniquement après l'affichage d'un message d'erreur. Parfois, on n'a pas de message d'erreur, mais le résultat retourné n'est pas celui attendu et il peut s'agir d'un problème de programmation. Dans ce cas là, l'idée est d'exécuter son code ligne après ligne et de vérifier les étapes intermédiaires.

On considère la fonction suivante :

```
ex_bug.2 <- function(x){
  x <- x + 1
  x <- x^3
  x <- log(x)
x
}
```

Pour utiliser la fonction `debugonce()` :

```
debugonce(ex_bug.2)
ex_bug.2(-5)
```

En exécutant pas à pas la fonction `ex_bug.2(-5)`, on peut demander à chaque étape la valeur que prend la variable `x` en cours d'exécution ; il faut pour cela saisir la variable `x` devant le prompt du débugger (`Browse[2]>`).

3.9.3.4 Négliger un message d'erreur

Une erreur entraîne l'interruption automatique du processus en cours d'exécution. Parfois, on peut avoir une erreur, mais on ne souhaite pas que cela interrompe le processus.

3.9.3.4.1 La fonction `try()` Si on connaît l'instruction qui peut entraîner une erreur, il est possible de lui appliquer la fonction `try()` :

```
f_error.1 <- function(x) {
  try(x <- log(x))
  x
}
```

Ci-dessous, on constate que le message d'erreur est affiché, mais cela n'a pas interrompu la fonction.

```
f_error.1("10")
## Error in log(x) : argument non numérique pour une fonction mathématique
```

```
## [1] "10"
```

Bibliographie : pour plus de d'informations sur le débogage, on renvoie le lecteur au chapitre “Debugging, condition handling, and defensive programming” du livre “Advanced R” d’Hadley Wickham.

3.10 Programmation orientée : S3/S4

3.10.1 La norme S3

Nous nous sommes inspirés du document suivant pour décrire la programmation orientée qui utilise la norme S3 : <http://www.duclert.org/r-divers/classes-S3-R.php>. Pour expliquer son principe, nous allons prendre un exemple concret.

On voudrait calculer l’aire de différentes figures géométriques :

- un carré, dont il nous faut connaître la longueur d’un côté, par exemple :

```
squ <- 3
```

- un rectangle, défini par sa longueur et sa largeur, par exemple :

```
rec <- c(5, 6)
```

- un cercle, caractérisé par son rayon, par exemple :

```
cir <- sqrt(10)
```

Pour ce faire, on souhaiterait utiliser une fonction qui s’appelle *getArea()* et qui puisse reconnaître quand on l’applique à un carré, un rectangle ou un cercle afin d’utiliser la bonne formule. Ceci consiste à définir une méthode générique **getArea** qui renvoie à la bonne implémentation en fonction de la classe de l’argument d’entrée. C’est ce qu’on appelle le polymorphisme.

3.10.1.1 Définition d’une nouvelle classe

Pour pouvoir être identifié comme étant un objet particulier, les trois objets ci-dessus doivent être définis comme appartenant à une certaine classe, que l’utilisateur, dans la norme S3, peut lui-même créer. On définit ainsi ci-dessous les trois types de classe **carre**, **rectangle** et **cercle** en utilisant la fonction *class()* :

```
class(cir) <- "cercle"
cir
```

```
## [1] 3.162278
## attr(,"class")
## [1] "cercle"

class(rec) <- "rectangle"
rec
```

```

## [1] 5 6
## attr(,"class")
## [1] "rectangle"
class(squ) <- "carre"
squ

## [1] 3
## attr(,"class")
## [1] "carre"

```

Remarque : à ce stade, nous n'avons créé aucune fonction qui puisse s'appliquer spécifiquement à ces objets. Cependant, comme ils ont été construits à la base comme des vecteurs, on peut toujours appliquer toutes les fonctions qui s'appliquent à des vecteurs de **numeric**.

Par ailleurs, pour calculer l'aire de ces objets, on pourrait écrire la fonction suivante qui traite le cas de chaque classe:

```

area <- function(x) {
  switch(class(x),
    carre = x ^ 2,
    rectangle = x[1] * x[2],
    cercle = pi * x ^ 2,
    "class should be among carre/rec/cercle")
}

```

Cependant, si on considère une nouvelle classe d'objet (un triangle par exemple), on devra reprogrammer la fonction *area()*.

C'est pourquoi on définit une méthode générique afin de pouvoir greffer autant de nouvelles classes d'objet sur celle-ci (par exemple, **plot** est une méthode sur laquelle on peut appliquer différents types d'objets).

3.10.1.2 Définition d'une méthode générique

On va créer la fonction *getArea()* qui va s'appliquer à ces classes d'objet en 3 étapes : **getArea** sera définie comme étant une méthode générique.

- La première étape consiste à définir la fonction *getArea()* comme étant une méthode générique. Autrement dit, on signale à **R** que cette fonction sera ensuite implémentée sous différentes versions :

```

getArea <- function(obj)
  UseMethod("getArea", obj)

```

- La deuxième étape consiste à créer une fonction par défaut qui sera appliquée dans le cas où on appliquerait à la fonction *getArea()* une classe d'objet autre que **carre**, **rectangle** ou **cercle**.

```
getArea.default <- function(obj) {
  stop("Méthode getArea non définie pour ce type d'objet")
}
```

- Enfin, on va créer les 3 “sous” fonctions de `getArea()` en ajoutant simplement à `getArea` le suffixe `.cercle`, `.rectangle` et `.carre` :

```
getArea.cercle <- function(obj) {
  pi * obj[1]^2
}

getArea.rectangle <- function(obj) {
  obj[1] * obj[2]
}

getArea.carre <- function(obj) {
  obj[1]^2
}
```

Voici un exemple d’application d’utilisation de la méthode `getArea` qu’on vient de créer :

```
getArea(cir)
## [1] 31.41593
getArea(rec)
## [1] 30
getArea(squ)
## [1] 9
```

Si on applique la méthode `getArea` à une classe d’objet non défini :

```
a <- 5
getArea(a)
```

```
## Error in getArea.default(a): Méthode getArea non définie pour ce type d'objet
```

Exercice 2.8

Ajouter à la méthode `getArea`, la classe triangle

3.10.1.3 Exemple de méthodes génériques

Il existe de nombreuses méthodes sous **R** qui sont déclarées comme génériques dans la norme S3. Par exemple, lorsqu’on utilise la fonction `plot()` ci-dessous, on l’applique dans le premier cas sur deux vecteurs **numeric** et dans le deuxième cas, on l’applique sur le résultat de la fonction `lm()` :

```
plot(rnorm(10), runif(10))
plot(lm(Sepal.Length ~ Sepal.Width, data = iris))
```

De même, pour savoir si un objet appartient à la norme S3, ce qui sous-entend que des méthodes génériques pourront s'appliquer sur cet objet, on peut utiliser la fonction `otype()` du package **pryr** :

```
library("pryr")

df <- data.frame(x = 1:10, y = letters[1:10])
pryr::otype(df)

## [1] "S3"
```

On constate qu'un **data.frame** appartient à la norme S3, ce qui sous entend qu'un nombre de méthodes génériques pourront s'appliquer dessus. La plupart du temps, ces objets qui appartiennent à la norme S3 sont créés à partir de **list**. Pour savoir comment ils sont constitués, on peut donc simplement utiliser la fonction `str()`.

3.10.1.3.1 Pour connaître les objets qui s'appliquent sur une méthode générique

On utilise la fonction `methods()` appliquée au nom de la méthode. Par exemple :

```
methods("plot")

## [1] plot,ANY,ANY-method
## [2] plot,color,ANY-method
## [3] plot,psi_func,ANY-method
## [4] plot,Spatial,missing-method
## [5] plot,SpatialGrid,missing-method
## [6] plot,SpatialDataFrame,missing-method
## [7] plot,SpatialLines,missing-method
## [8] plot,SpatialMultiPoints,missing-method
## [9] plot,SpatialPixels,missing-method
## [10] plot,SpatialPixelsDataFrame,missing-method
## [11] plot,SpatialPoints,missing-method
## [12] plot,SpatialPolygons,missing-method
## [13] plot.acf*
## [14] plot.aggr*
## [15] plot.amelia*
## [16] plot.bclust*
## [17] plot.boot*
## [18] plot.classIntervals*
## [19] plot.correspondence*
## [20] plot.data.frame*
## [21] plot.decomposed.ts*
```

```
## [22] plot.default
## [23] plot.dendrogram*
## [24] plot.density*
## [25] plot.ecdf
## [26] plot.factor*
## [27] plot.formula*
## [28] plot.function
## [29] plot.ggplot*
## [30] plot.goodfit*
## [31] plot.gtable*
## [32] plot.hcl_palettes*
## [33] plot.hclust*
## [34] plot.histogram*
## [35] plot.HoltWinters*
## [36] plot.ica*
## [37] plot.isoreg*
## [38] plot.lda*
## [39] plot.lm*
## [40] plot.lmrob*
## [41] plot.loadings*
## [42] plot.loddsratio*
## [43] plot.loglm*
## [44] plot.lowess*
## [45] plot.lts*
## [46] plot.margin*
## [47] plot.mca*
## [48] plot.mcd*
## [49] plot.medpolish*
## [50] plot.mlm*
## [51] plot.mvr*
## [52] plot.mvrVal*
## [53] plot.numpy.ndarray*
## [54] plot.paretoTail*
## [55] plot.ppr*
## [56] plot.prcomp*
## [57] plot.princomp*
## [58] plot.profile*
## [59] plot.profile.nls*
## [60] plot.R6*
## [61] plot.randomForest*
## [62] plot.raster*
## [63] plot.ridgelm*
## [64] plot.scores*
## [65] plot.shingle*
## [66] plot.SOM*
## [67] plot.somgrid*
```

```

## [68] plot.spec*
## [69] plot.stepfun
## [70] plot.stft*
## [71] plot.stl*
## [72] plot.structure*
## [73] plot.svm*
## [74] plot.table*
## [75] plot.trans*
## [76] plot.trellis*
## [77] plot.ts
## [78] plot.tskernel*
## [79] plot.TukeyHSD*
## [80] plot.tune*
## [81] plot.venn
## [82] plot.zoo
## see '?methods' for accessing help and source code

```

3.10.1.3.2 Pour connaître les méthodes qui s’appliquent sur une classe d’objet On utilise la fonction *methods()* en précisant l’argument **class** =. Par exemple :

```

methods(class = "lm")

## [1] add1      alias      anova      case.names coerce
## [6] confint   cooks.distance deviance  dfbeta    dfbetas
## [11] drop1     dummy.coef   effects    extractAIC family
## [16] formula   fortify    hatvalues  influence  initialize
## [21] kappa     labels     logLik     model.frame model.matrix
## [26] nobs      plot       predict    print      proj
## [31] qr        residuals  rstandard  rstudent   show
## [36] simulate  slotsFromS3 summary   variable.names vcov
## see '?methods' for accessing help and source code

```

Pour en savoir plus : le lecteur pourra consulter la dernière partie du lien suivant qui donne un exemple de création de méthode générique plus complet, avec la possibilité de définir les commandes +, [], etc. sur de nouvelles classes d’objet : <http://www.duclert.org/r-divers/classes-S3-R.php>.

3.10.2 La norme S4

Ici, on présentera uniquement les grandes lignes de la norme S4. Cette section est inspirée du chapitre OO field guide du livre d’Hadley Wickham.

La norme S4 s’inspire de la norme S3 auxquelles s’ajoutent certaines caractéristiques telles que :

- les classes sont définies en ajoutant un certain nombre de règles et de précisions les concernant. Pour mieux comprendre, dans l’exemple ci-dessous,

on peut définir l'objet **a** comme appartenant à la classe **lm** dans la norme S3, alors qu'il n'en possède pas les caractéristiques :

```
a <- 5
class(a) <- "lm"
```

- on utilisera un opérateur spécial **@** pour extraire des éléments d'un objet ayant la norme S4.

Pour savoir si un objet appartient à la norme S4, on peut utiliser la fonction *isS4()*. La plupart du temps, la norme S4 n'a pas été utilisée pour définir des objets et fonctions qui appartiennent à l'environnement de base. Ils sont donc plutôt présents dans des nouveaux packages qu'on va charger.

Les principales fonctions utilisées pour travailler sur la norme S4 sont :

- la fonction *setClass()* qui permet de définir une nouvelle classe. Le 1er argument est le nom de la classe, le second est le résultat de la fonction *representation()* qui définit le nom et le type des éléments que la classe contient et le troisième argument est le résultat de la fonction *prototype()* qui donne des valeurs par défaut. Par exemple :

```
setClass("Personne", representation(nom = "character", age = "numeric"),
         prototype(nom = NA_character_, age = NA_real_))
```

- la fonction *new()* permet de créer un nouvel objet.

```
hadley <- new("Personne", nom = "Hadley", age = 31)
```

Remarque : on accède aux éléments d'un objet de norme S4 avec la commande spéciale **@** :

```
hadley@age
```

```
## [1] 31
```

Les autres fonctions utiles dont vous trouverez des exemples d'utilisation dans <http://adv-r.had.co.nz/S4.html> sont :

- la fonction *setMethods()* permettant de définir des nouvelles méthodes.
- les fonctions *as()* et *setAs()* pour passer d'une classe d'objet à une autre (quand cela est possible).
- les fonctions *setValidity()* et *validObject()* pour vérifier la validité.
- les fonctions *showClass()*, *showMethods()* et *getMethod()* pour accéder aux propriétés des objets et méthodes créées.

Bibliographie : la présentation de F. Leisch à useR! 2004 (<http://www.ci.tuwien.ac.at/Conferences/useR-2004/Keynotes/Leisch.pdf>) ainsi que le manuel de C. Genolini (<https://cran.r-project.org/doc/contrib/Genolini-PetitManuelDeS4.pdf>).

3.11 Visualiser le code source d'une fonction

Nous nous sommes inspirés du document suivant pour écrire cette section : “Visualiser le code d'une fonction”.

3.11.1 La fonction est dans l'environnement courant

Vous tapez le nom de la fonction dans la console et le code apparaît :

```
sapply
```

```
## function (X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
## {
##   FUN <- match.fun(FUN)
##   answer <- lapply(X = X, FUN = FUN, ...)
##   if (USE.NAMES && is.character(X) && is.null(names(answer)))
##     names(answer) <- X
##   if (!isFALSE(simplify))
##     simplify2array(answer, higher = (simplify == "array"))
##   else answer
## }
## <bytecode: 0x55cc3db093e8>
## <environment: namespace:base>
```

3.11.2 La fonction est une méthode générique de type S3

Vous tapez le nom d'une fonction *lambda()* et vous obtenez une ligne **UseMethod("lambda")**. Ceci signifie que **lambda** est une méthode générique pour des objets issus de la classe S3. Il existe donc plusieurs fonctions associées à la fonction *lambda()*. Pour connaître quelles sont les différentes fonctions associées à la fonction *lambda()*, on utilise la fonction *methods()*. Par exemple, pour la fonction *summary()*, si on tape le nom de la fonction dans la console, on obtient :

```
summary
```

```
## function (object, ...)
## UseMethod("summary")
## <bytecode: 0x55cc41917e70>
## <environment: namespace:base>
```

Pour obtenir les différentes fonctions associées à *summary()*, on utilise donc la fonction *methods()*.

```
methods("summary")
```

| | |
|--|--|
| <pre>## [1] summary,ANY-method ## [3] summary,diagonalMatrix-method ## [5] summary,sparseMatrix-method</pre> | <pre>summary,DBIObject-method summary,GridTopology-method summary,Spatial-method</pre> |
|--|--|

```

## [7] summary.aggr*
## [9] summary.Anova.mlm*
## [11] summary.aovlist*
## [13] summary.assocstats*
## [15] summary.bcnPowerTransformlmer*
## [17] summary.check_packages_in_dir*
## [19] summary.data.frame
## [21] summary.default
## [23] summary.ecdf*
## [25] summary.ggplot*
## [27] summary.glmrob*
## [29] summary.haven_labelled*
## [31] summary.infl*
## [33] summary.Kappa*
## [35] summary.lm
## [37] summary.lodds*
## [39] summary.loess*
## [41] summary.lts*
## [43] summary.matrix
## [45] summary.mi*
## [47] summary.mlm*
## [49] summary.mvr*
## [51] summary.nlrob*
## [53] summary.nnet*
## [55] summary.pandas.core.frame.DataFrame*
## [57] summary.Period*
## [59] summary.POSIXct
## [61] summary.powerTransform*
## [63] summary.pr_DB*
## [65] summary.princomp*
## [67] summary.proxy_registry*
## [69] summary.rlang_error*
## [71] summary.rlm*
## [73] summary.srcfile
## [75] summary.stepfun
## [77] summary.svm*
## [79] summary.trellis*
## [81] summary.tune*
## [83] summary.vctrs_vctr*
## [85] summary.yearmon*
## [87] summary.zoo*
## see '?methods' for accessing help and source code

```

Pour lire le code des fonctions affichées ci-dessus, il y a deux options :

- si il n'y pas d'étoile à côté de la fonction, vous pouvez obtenir directement le code de la fonction en tapant son nom complet :

```
summary.proc_time
```

```
## function (object, ...)
## {
##   if (!is.na(object[4L]))
##     object[1L] <- object[1L] + object[4L]
##   if (!is.na(object[5L]))
##     object[2L] <- object[2L] + object[5L]
##   object <- object[1L:3L]
##   names(object) <- cgettext("user"), gettext("system"), gettext("elapsed"))
##   object
## }
## <bytecode: 0x55cc41918298>
## <environment: namespace:base>
```

- s'il y a une étoile, c'est que la fonction se trouve dans un package et qu'on doit indiquer le nom du package suivi de :: :

```
stats:::summary.ecdf
```

Si on ne connaît pas le nom du package, on peut alors utiliser la fonction *getAnywhere()* qui va chercher dans toutes les librairies chargées :

```
getAnywhere(summary.ecdf)
```

```
## A single object matching 'summary.ecdf' was found
## It was found in the following places
##   registered S3 method for summary from namespace stats
##   namespace:stats
##   with value
##
## function (object, ...)
## {
##   n <- length(eval(expression(x), envir = environment(object)))
##   header <- paste("Empirical CDF:\t ", n, "unique values with summary\n")
##   structure(summary(knots(object)), ...), header = header, class = "summary.ecdf")
## }
## <bytecode: 0x55cc5b7460d8>
## <environment: namespace:stats>
```

3.11.3 La fonction fait appel à du code C

Si on affiche le code la fonction *sum()*, on constate que le calcul n'est pas directement fait dans le corps de la fonction. C'est ce que nous avons vu précédemment, à savoir qu'il existe un grand nombre de fonctions qui font appel à du code compilé pour gagner en performances. Lorsqu'on voit dans une fonction les instructions **.Primitive()** ou **.Internal()**, cela signifie que R va en réalité

exécuter des fonctions qui ont été compilées en **C**. La différence entre **.Primitive()** ou **.Internal()** vient du fait que dans le deuxième cas, on peut appeler des fonctions qui sont codées en langage **R**.

```
sum
```

```
## function (..., na.rm = FALSE) .Primitive("sum")
```

On peut donc faire appel à du code **C** de plusieurs façons. On verra ici les deux principales façons de le faire.

3.11.3.1 .Primitive() ou .Internal()

Il est difficile d'accéder directement au code source de la fonction exécutée par **.Primitive()** ou **.Internal()**. Il s'agit des fonctions qui font en général partie du package **base**. On peut utiliser la fonction *show_c_source()* du package **pryr** qui va effectuer une recherche sur github où a été déposé le source de l'ensemble des fichiers de codes pré-compilés utilisés dans **R** :

```
pryr::show_c_source(.Internal(mean(x)))
```

3.11.3.2 .Call()

Dans d'autres situations, une fonction fait appel à du code **C** via la fonction **.Call()**. Par exemple :

```
qnorm
```

```
## function (p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
## .Call(C_qnorm, p, mean, sd, lower.tail, log.p)
## <bytecode: 0x55cc5bba54e8>
## <environment: namespace:stats>
```

A priori, le code source est disponible dans les répertoires nommés **src** : soit dans **R**, soit dans les packages installés (dans les deux cas, les codes sources doivent donc être accessibles depuis les répertoires). Sinon, on peut faire une recherche en ligne en tapant dans le moteur de recherche les instructions suivantes :

```
site:https://svn.r-project.org/R/trunk/src qnorm
```

Bibliographie : pour en savoir plus sur le langage compilé, on pourra lire le chapitre R's C interface du livre d'Hadley Wickham, "Advanced R".

3.11.4 La fonction est une méthode générique de type S4

C'est le cas des fonctions programmées dans le package **Matrix**.

```
require("Matrix")
```

Si on utilise la fonction `methods()` sur la méthode `dim`, pour identifier les méthodes génériques qui s'appliquent à des objets S4, on constate que le nom de la fonction est suivi d'une virgule, suivi alors de la classe de l'objet sur laquelle s'applique la méthode. Dans le cas de `dim()`, elle peut s'appliquer sur les classes d'objet **MatrixFactorization** ou **Matrix**.

```
methods("dim")

## [1] dim,Matrix-method           dim,MatrixFactorization-method
## [3] dim.cell_limits*           dim.data.frame
## [5] dim.data.table*            dim.dist*
## [7] dim.gtable*                dim.layout*
## [9] dim.lodds*                 dim.loddsratio*
## [11] dim.pandas.core.frame.DataFrame* dim.pandas.core.series.Series*
## [13] dim.permutation*          dim.resample*
## [15] dim.scipy.sparse.base.spmatrix* dim.SpatialGridDataFrame*
## [17] dim.SpatialLinesDataFrame* dim.SpatialMultiPointsDataFrame*
## [19] dim.SpatialPixelsDataFrame* dim.SpatialPointsDataFrame*
## [21] dim.SpatialPolygonsDataFrame* dim.structable*
## [23] dim.tbl_lazy*              dim.trellis*
## see '?methods' for accessing help and source code
```

Pour obtenir le code source de la fonction `dim()` qui s'applique à la classe d'objet **MatrixFactorization**, on utilise la fonction `getMethod()` ainsi :

```
getMethod("dim", "MatrixFactorization")
```

```
## Method Definition:
##
## function (x)
## x@Dim
## <bytecode: 0x55cc52cfac68>
## <environment: namespace:Matrix>
##
## Signatures:
##   x
## target  "MatrixFactorization"
## defined "MatrixFactorization"
```

Pour connaître les méthodes S4 qui s'appliquent sur une classe d'objets particuliers, on utilise la fonction `showMethods()` :

```
showMethods(class = "MatrixFactorization")
```

```
##
## Function "%m-%":
## <not an S4 generic function>
##
## Function "%m+%" :
```

```
## <not an S4 generic function>
##
## Function "%within%":
## <not an S4 generic function>
##
## Function "addAttrToGeom":
## <not an S4 generic function>
##
## Function "as_date":
## <not an S4 generic function>
##
## Function "as_datetime":
## <not an S4 generic function>
##
## Function "as.duration":
## <not an S4 generic function>
##
## Function "as.interval":
## <not an S4 generic function>
##
## Function "as.period":
## <not an S4 generic function>
##
## Function "asJSON":
## <not an S4 generic function>
##
## Function "bbox":
## <not an S4 generic function>
##
## Function "chgDefaults":
## <not an S4 generic function>
##
## Function "complete":
## <not an S4 generic function>
##
## Function "coordinates":
## <not an S4 generic function>
##
## Function "coordinates<-":
## <not an S4 generic function>
##
## Function "coordnames":
## <not an S4 generic function>
```

```
##  
## Function "coordnames<-" :  
## <not an S4 generic function>  
##  
## Function "date<-" :  
## <not an S4 generic function>  
##  
## Function "day<-" :  
## <not an S4 generic function>  
##  
## Function "dbAppendTable" :  
## <not an S4 generic function>  
##  
## Function "dbBegin" :  
## <not an S4 generic function>  
##  
## Function "dbBind" :  
## <not an S4 generic function>  
##  
## Function "dbCallProc" :  
## <not an S4 generic function>  
##  
## Function "dbCanConnect" :  
## <not an S4 generic function>  
##  
## Function "dbClearResult" :  
## <not an S4 generic function>  
##  
## Function "dbColumnInfo" :  
## <not an S4 generic function>  
##  
## Function "dbCommit" :  
## <not an S4 generic function>  
##  
## Function "dbConnect" :  
## <not an S4 generic function>  
##  
## Function "dbCreateTable" :  
## <not an S4 generic function>  
##  
## Function "dbDataType" :  
## <not an S4 generic function>  
##  
## Function "dbDisconnect" :  
## <not an S4 generic function>  
##
```

```
## Function "dbDriver":  
## <not an S4 generic function>  
##  
## Function "dbExecute":  
## <not an S4 generic function>  
##  
## Function "dbExistsTable":  
## <not an S4 generic function>  
##  
## Function "dbFetch":  
## <not an S4 generic function>  
##  
## Function "dbGetConnectArgs":  
## <not an S4 generic function>  
##  
## Function "dbGetException":  
## <not an S4 generic function>  
##  
## Function "dbGetInfo":  
## <not an S4 generic function>  
##  
## Function "dbGetQuery":  
## <not an S4 generic function>  
##  
## Function "dbGetRowCount":  
## <not an S4 generic function>  
##  
## Function "dbGetRowsAffected":  
## <not an S4 generic function>  
##  
## Function "dbGetStatement":  
## <not an S4 generic function>  
##  
## Function "dbHasCompleted":  
## <not an S4 generic function>  
##  
## Function "dbiDataType":  
## <not an S4 generic function>  
##  
## Function "dbIsReadOnly":  
## <not an S4 generic function>  
##  
## Function "dbIsValid":  
## <not an S4 generic function>  
##  
## Function "dbListConnections":
```

```
## <not an S4 generic function>
##
## Function "dbListFields":
## <not an S4 generic function>
##
## Function "dbListObjects":
## <not an S4 generic function>
##
## Function "dbListResults":
## <not an S4 generic function>
##
## Function "dbListTables":
## <not an S4 generic function>
##
## Function "dbQuoteIdentifier":
## <not an S4 generic function>
##
## Function "dbQuoteLiteral":
## <not an S4 generic function>
##
## Function "dbQuoteString":
## <not an S4 generic function>
##
## Function "dbReadTable":
## <not an S4 generic function>
##
## Function "dbRemoveTable":
## <not an S4 generic function>
##
## Function "dbRollback":
## <not an S4 generic function>
##
## Function "dbSendQuery":
## <not an S4 generic function>
##
## Function "dbSendStatement":
## <not an S4 generic function>
##
## Function "dbSetDataMappings":
## <not an S4 generic function>
##
## Function "dbUnloadDriver":
## <not an S4 generic function>
##
## Function "dbUnquoteIdentifier":
## <not an S4 generic function>
```

```
##  
## Function "dbWithTransaction":  
## <not an S4 generic function>  
##  
## Function "dbWriteTable":  
## <not an S4 generic function>  
## Function: dim (package base)  
## x="MatrixFactorization"  
##  
##  
## Function "dimensions":  
## <not an S4 generic function>  
##  
## Function "disaggregate":  
## <not an S4 generic function>  
## Function: expand (package Matrix)  
## x="MatrixFactorization"  
##  
##  
## Function "fetch":  
## <not an S4 generic function>  
##  
## Function "format_ISO8601":  
## <not an S4 generic function>  
##  
## Function "fullgrid":  
## <not an S4 generic function>  
##  
## Function "fullgrid<-":  
## <not an S4 generic function>  
##  
## Function "functions":  
## <not an S4 generic function>  
##  
## Function "geometry":  
## <not an S4 generic function>  
##  
## Function "geometry<-":  
## <not an S4 generic function>  
##  
## Function "gridded":  
## <not an S4 generic function>  
##  
## Function "gridded<-":  
## <not an S4 generic function>  
##
```

```
## Function "hour<-" :
## <not an S4 generic function>
##
## Function "is.projected" :
## <not an S4 generic function>
##
## Function "isSQLKeyword" :
## <not an S4 generic function>
##
## Function "make.db.names" :
## <not an S4 generic function>
##
## Function "merge" :
## <not an S4 generic function>
##
## Function "minute<-" :
## <not an S4 generic function>
##
## Function "month<-" :
## <not an S4 generic function>
##
## Function "over" :
## <not an S4 generic function>
##
## Function "polygons" :
## <not an S4 generic function>
##
## Function "polygons<-" :
## <not an S4 generic function>
##
## Function "proj4string" :
## <not an S4 generic function>
##
## Function "proj4string<-" :
## <not an S4 generic function>
##
## Function "qday<-" :
## <not an S4 generic function>
##
## Function "rebuild_CRS" :
## <not an S4 generic function>
##
## Function "recenter" :
## <not an S4 generic function>
##
## Function "reclass_timespan" :
```

```
## <not an S4 generic function>
##
## Function "second<-":
## <not an S4 generic function>
## Function: show (package methods)
## object="MatrixFactorization"
##
## Function: solve (package base)
## a="MatrixFactorization", b="ANY"
## a="MatrixFactorization", b="missing"
## a="MatrixFactorization", b="numeric"
##
##
## Function "spChFIDs":
## <not an S4 generic function>
##
## Function "spChFIDs<-":
## <not an S4 generic function>
##
## Function "split":
## <not an S4 generic function>
##
## Function "sppanel":
## <not an S4 generic function>
##
## Function "spplot":
## <not an S4 generic function>
##
## Function "spsample":
## <not an S4 generic function>
##
## Function "spTransform":
## <not an S4 generic function>
##
## Function "sqlAppendTable":
## <not an S4 generic function>
##
## Function "sqlCreateTable":
## <not an S4 generic function>
##
## Function "sqlData":
## <not an S4 generic function>
##
## Function "sqlInterpolate":
## <not an S4 generic function>
##
```

```
## Function "SQLKeywords":  
## <not an S4 generic function>  
##  
## Function "sqlParseVariables":  
## <not an S4 generic function>  
##  
## Function "surfaceArea":  
## <not an S4 generic function>  
##  
## Function "time_length":  
## <not an S4 generic function>  
##  
## Function "wkt":  
## <not an S4 generic function>  
##  
## Function "year<-":  
## <not an S4 generic function>
```


Chapter 4

Calcul parallèle

Nous allons voir dans ce chapitre le principe du calcul parallèle et une façon d'en faire avec **R**. Ce document est fortement inspiré du tutoriel suivant Introduction to parallel computing with R. Le lecteur pourra également consulter cette présentation très intéressante de Vicent Miele Le calcul parallèle pour non-spécialistes,c'est maintenant!

Ce document a été généré directement depuis **RStudio** en utilisant l'outil Markdown. La version *.pdf* se trouve ici.

Packages à installer

```
install.packages(c("parallel", "snow", "snowFT", "VGAM"),
                  dependencies = TRUE)
```

4.1 Principe du calcul parallèle

On suppose que nous ayons un calcul à réaliser qui aurait cette forme :

```
initialize.rng(...) # on définit une séquence de graines pour tirage aléatoire
for (iteration in 1:N) { # on répète N fois,
  result[iteration] <- myfunc(...) # la fonction myfunc()
  # avec une valeur de graine différente par itération
}
process(result,...) # récupération des résultats
```

Exemple: un exemple de programme ayant cette forme serait l'algorithme des forêts aléatoires, dans lequel *myfunc()* permettrait de coder un arbre de régression ou de classification sur un échantillon qui serait tiré différemment à chaque itération. A la fin de la boucle, on agrège en général les résultats des différents arbres pour faire de la prédiction.

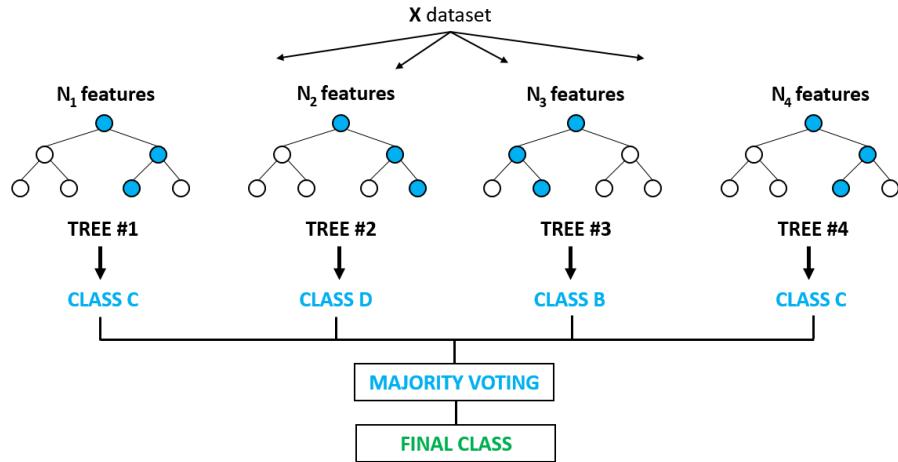


Figure 4.1: Schéma des forêts aléatoires

Si on ne précise rien, le calcul précédent sera effectué de façon séquentiel, autrement dit il faut attendre qu'une itération soit terminée pour passer à la suivante. L'idée du calcul parallèle est de permettre de lancer le calcul de la fonction *myfunc()*, en parallèle, comme le montre la figure suivante.

Avant de continuer à parler de calcul parallèle, rappelons la définition d'un processeur : “Le processeur ou CPU (Central Processing Unit) est le composant de votre ordinateur qui exécute les instructions qui lui sont données par votre système d’exploitation. Quand vous exécutez un logiciel, décompressez une archive ZIP ou regardez une vidéo en haute définition, vous faites travailler en priorité le processeur ! Pour répondre à vos demandes les plus exigeantes, le processeur peut être doté de plusieurs coeurs.” (définition extraite de ce site)

Si un processeur ne possède qu'un seul cœur, il ne sera pas possible faire du calcul parallèle car les instructions seront traitées en série. Aujourd’hui, la plupart des machines sont dotées d'un processeur multi cœur, composé de deux ou plusieurs coeurs indépendants. Un processeur dual-core contient deux coeurs, un processeur quad-core quatre coeurs, etc. Sur la représentation graphique ci-dessous, on distingue bien les quatre coeurs du processeur.

Pour savoir combien de coeurs on dispose sur notre machine, on peut utiliser la fonction *detectCores()* du package **parallel**:

```
library("parallel")
detectCores() # nombre de coeurs physiques
## [1] 40
```

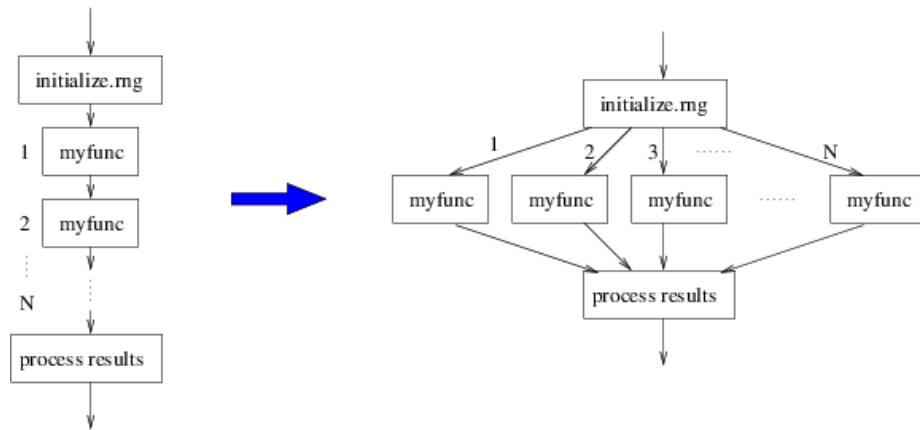


Figure 4.2: Exemple de calcul parallèle

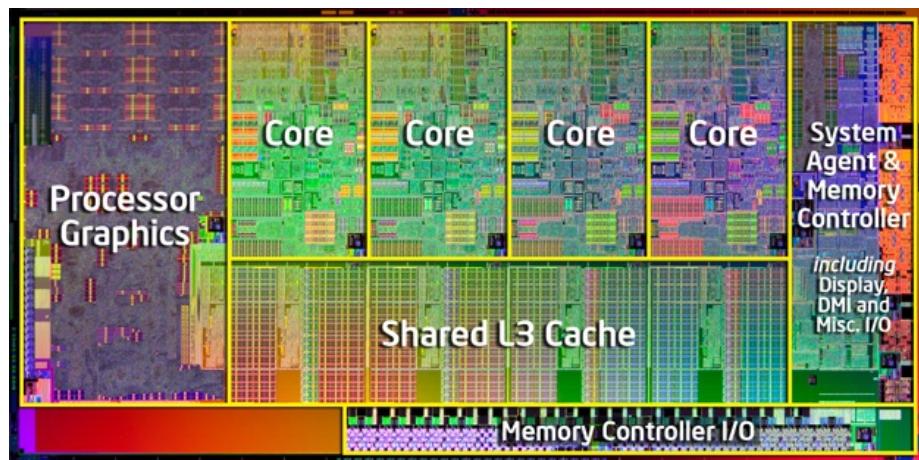


Figure 4.3: CPU

Cependant, un coeur peut lui-même se dédoubler : le nombre de tâches pouvant être exécuté correspond au nombre de “Thread” ou processeurs logiques. Pour connaître le nombre de Threads, on exécute la commande suivante :

```
detectCores(logical = TRUE) # nombre de Threads
```

```
## [1] 40
```

Sur Windows, il y a en général deux fois plus de Threads que de coeurs physiques. Dans notre cas, c'est le nombre de Threads qui nous intéresse plus particulièrement et c'est ce dernier chiffre que nous utiliserons pour faire du calcul parallèle. Dans la suite du cours, j'utiliserai l'expression coeur pour définir un Thread.

Pour en savoir plus sur les définitions des coeurs logiques et physiques, vous pouvez consulter cette page wikipedia

4.2 Notion de programme maître

Dans l'exemple précédent, on a vu qu'on souhaitait répliquer N fois la fonction *myfunc()*, sachant que pour chaque itération, on utilise une valeur de graine différente (ce qui est logique sinon cela voudrait dire qu'on aurait N fois le même résultat). Si N est plus grand que le nombre de coeurs dont on dispose (ce qui est le plus souvent le cas), l'idée sera d'envoyer sur chaque coeur un certain nombre d'itération. Prenons par exemple $N = 100$ et 4 coeurs, dans ce cas il semble naturel de répartir les tâches de la façon suivante :

- le coeur 1 lancera les itérations 1, 5, 9, ... 93 et 97.
- le coeur 2 lancera les itérations 2, 6, 10, ..., 94 et 98.
- le coeur 3 lancera les itérations 3, 7, 11, ..., 95 et 99.
- le coeur 4 lancera les itérations 4, 8, 12, ..., 96 et 100

L'idée du programme maître est qu'il devra spécifier cette répartition des tâches. Autrement dit, il devra indiquer que la fonction *myfunc()* exécutera les itérations 1, 5, 9, ... 93 et 97 dans le coeur 1, les itérations 2, 6, 10, ..., 94 et 98 dans le coeur 2, etc.

Une fois les tâches effectuées, l'autre rôle du programme maître est de bien récupérer les sorties du programme parallélisé et d'en faire en général une synthèse (calcul de moyenne, d'écart-types, etc.).

On peut résumer ceci par la figure suivante :

Il existe énormément de package permettant de faire du calcul parallèle. Nous utiliserons ici les packages **parallel** et **snow**.

4.3 Fonction *set.seed()*

Quand on fait de l'échantillonage ou qu'on simule des échantillons de façon aléatoire, on peut avoir besoin de retrouver les mêmes tirages. C'est le cas

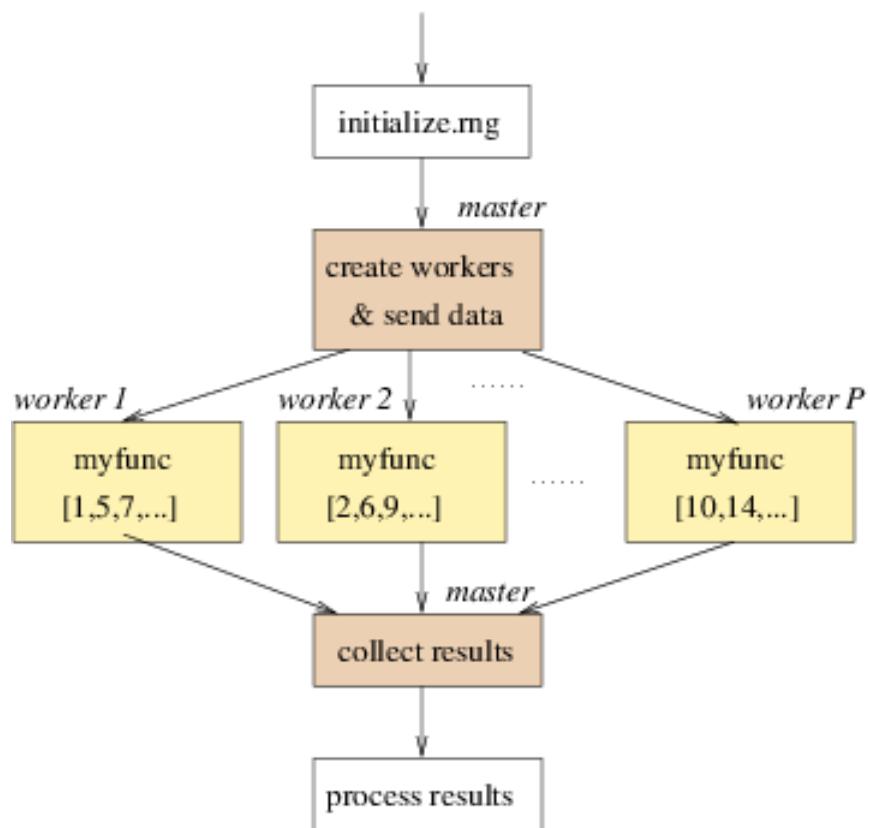


Figure 4.4: Exemple de programme maître

notamment lorsqu'on présente des résultats dans un rapport et qu'on souhaite que les résultats soient reproductibles.

La fonction `set.seed()` est utilisée pour reproduire les mêmes tirages/échantillonages les uns à la suite des autres. Elle prend comme argument d'entrée un entier appelé **graine** ou “seed” qui est utilisée dans l'algorithme de simulation. En effet, la notion d'aléatoire au sens strict n'existe pas lorsqu'on fait de la simulation numérique. Dans la vraie vie, lorsqu'on tire une boule dans une urne (par exemple au tirage du loto), on comprend bien qu'il ne sera pas possible de tirer les mêmes boules deux fois de suite (si la condition d'aléatoire est bien respectée). En revanche, avec des simulations numériques, comme on utilise un algorithme dit pseudo-aléatoire, cela devient possible de répliquer un même tirage en utilisant le même nombre initial dans l'algorithme.

La fonction `set.seed()` précède la fonction `sample()` ou tout autre fonction génératrice de lois de distribution connues (`rnorm()`, `runif()`, `rbinom()`, etc.). Par exemple, on souhaite faire à la suite :

- un tirage aléatoire de 5 nombres dans une urne comprenant 49 boules avec des numéros allant de 1 à 49 exemple,
- simuler un bruit blanc de taille 5

```
set.seed(200907)
(loto <- sample(1:49, 5))

## [1] 4 21 18 8 37
:white_noise <- rnorm(5)

## [1] 0.69019477 -0.13529509 0.69372207 -0.73089395 0.02133644
```

Si on répète cette même syntaxe, on obtiendra systématiquement les mêmes résultats.

```
set.seed(200907)
(loto <- sample(1:49, 5))

## [1] 4 21 18 8 37
:white_noise <- rnorm(5)

## [1] 0.69019477 -0.13529509 0.69372207 -0.73089395 0.02133644
```

Le lecteur trouvera plus d'informations sur la fonction `set.seed()` dans cette vidéo : <https://www.youtube.com/watch?v=zAYzAZwufKI>

Application dans un algorithme de type “bootstrap”

Dans le graphique suivant, on représente l'intérêt du bootstrap.

Pour que cela fonctionne, il faut qu'à chaque simulation, on tire un échantillon différent. Autrement dit, il faut veiller à ne pas appliquer la fonction `set.seed()` à un nombre constant dans la boucle **for**. Dans l'exemple suivant, on calcule

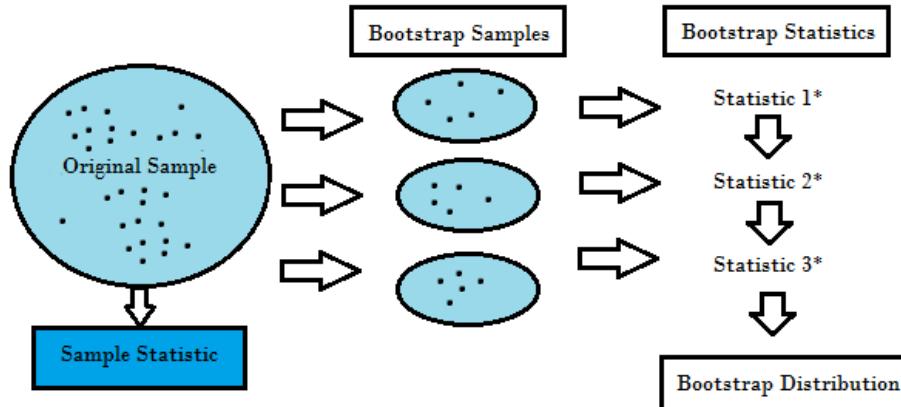


Figure 4.5: Exemple d'algorithme de type bootstrap

l'estimateur “bootstrap” de la moyenne de la variable **Sepal.Length** du jeu de données **iris**. Comme la graine est fixée, on va toujours tirer le même échantillon et on aura un estimateur biaisée.

```

B <- 10
res_mean <- numeric(B)
for (b in 1:B) {
  set.seed(123)
  samp <- sample(1:nrow(iris), replace = T)
  res_mean[b] <- mean(iris[samp, "Sepal.Length"])
}
res_mean

## [1] 5.774 5.774 5.774 5.774 5.774 5.774 5.774 5.774 5.774 5.774
  
```

Solution: pour corriger le programme ci-dessus, il suffit de remplacer la valeur de la graine 123 par l'objet **b** qui varie de 1 à B.

Exercice 3.1

On considère le modèle de régression suivant où les données **wage1** sont issues du package **wooldridge**:

$$\log(wage) = \beta_0 + \beta_1 educ + \epsilon$$

```

library(wooldridge)
log_wage_model <- lm(lwage ~ educ, data = wage1)
  
```

- Calculer un estimateur bootstrap de β_0 et β_1 . On fixe le nombre de répliques à $B = 1000$.
- Representer l'histogramme des valeurs bootstrappées (un histogramme par coefficient), l'estimateur “bootstrap” de la moyenne et la vraie valeur.
- Ecrire l'algorithme qui permette de faire la même chose en utilisant le calcul parallèle

4.4 Syntaxe pour lancer un calcul parallèle

On considère la fonction suivante qui permet de calculer la moyenne d'un échantillon de taille **r** simulée selon une loi normale de paramètre **mean** et **sd**.

```
myfun <- function(r, mean = 0, sd = 1) {
  mean(rnorm(r, mean = mean, sd = sd))
}
```

On souhaite répéter 100 fois cette fonction :

- 25 fois quand **r** vaut 10,
- 25 fois quand **r** vaut 1000,
- 25 fois quand **r** vaut 100000,
- 25 fois quand **r** vaut 1000000,

avec **mean=5** et **sd=10**.

4.4.1 Syntaxe dans le cas non parallèle

Pour répondre à la problématique posée, on va d'abord utiliser la fonction *sapply()* qui permet de répondre au problème de façon séquentielle. Autrement dit, elle va appliquer la fonction *myfun()* itération après itération sur les différentes valeurs de **r** qu'on donne dans le 1er argument de la fonction *sapply()*. D'abord, on crée le vecteur contenant les valeurs de **r** et on prépare aussi l'objet qui va contenir les résultats.

```
r_values <- rep(c(10, 1000, 100000, 10000000), each = 25)
resultats <- data.frame(r_values = factor(r_values))
```

On lance la fonction *sapply()* et on regarde le temps de calcul :

```
time_sapply <- system.time(
  resultats$res_non_par <- sapply(r_values, FUN = myfun,
    mean = 5, sd = 10) # options de la fonction myfun
)
time_sapply

## utilisateur      système     écoulé
##       15.890       0.832     16.723
```

4.4.2 Syntaxe dans le cas parallèle

Pour exécuter la fonction précédente dans le cas parallèle, la syntaxe est la suivante :

```
P <- 4 # définir le nombre de coeurs
cl <- makeCluster(P) # réserve 4 coeurs - début du calcul
time_par <- system.time(
  res_par <- clusterApply(cl, r_values, fun = myfun, # évalue myfun sur r_values
                          mean = 5, sd = 10) # options de myfun
)
stopCluster(cl) # libère 4 coeurs - fin du calcul
time_par

## utilisateur      système      écoulé
##       0.036        0.001      6.175
```

La syntaxe est donc pratiquement la même que dans le cas non parallèle. Il suffit simplement d'utiliser la fonction *makeCluster()* au début pour réserver le nombre de coeurs nécessaires et la fonction *stopCluster()* à la fin pour libérer les coeurs. De même, c'est la fonction *clusterApply()* qui permet de répartir la fonction *myfun()* vers les différents coeurs.

Pendant l'exécution d'un programme en parallèle, il est possible de vérifier que les coeurs sont en train de fonctionner en parallèle. Sur Windows, il suffit de cliquer sur le “Gestionnaire des tâches” pour voir l'état de l'utilisation des processeurs et/ou de la mémoire.

4.4.3 Recommandations

4.4.3.1 Gestion de la mémoire

Si vous parallélisez un programme qui utilise x Go de RAM, vous allez a priori avoir besoin de Px Go de RAM où P est le nombre de coeurs utilisés. Si dans le gestionnaire des tâches, vous vous rendez compte que vous utilisez un programme qui utilise 100% de la mémoire RAM, cela aura pour effet de réduire considérablement les temps de calcul.

4.4.3.2 En cas d'interruption du programme maître

Si vous quittez votre programme maître alors que le calcul parallèle est en train de tourner, dans ce cas il est probable que les coeurs alloués continuent de tourner même si votre session maître semble terminer (ceci peut se voir dans le “Gestionnaire des tâches” et “Processus”, plusieurs processus **R** seront ouverts). Dans ce cas, il faudra penser à exécuter la fonction *stopCluster()*. Si cela n'est pas suffisant (les coeurs continuent à tourner), il faudra vraisemblablement tuer les process à la main (dans “Gestionnaire des tâches”, puis “Processus”, puis click droit sur les icônes “R for Windows” et “Fin de tâche”).

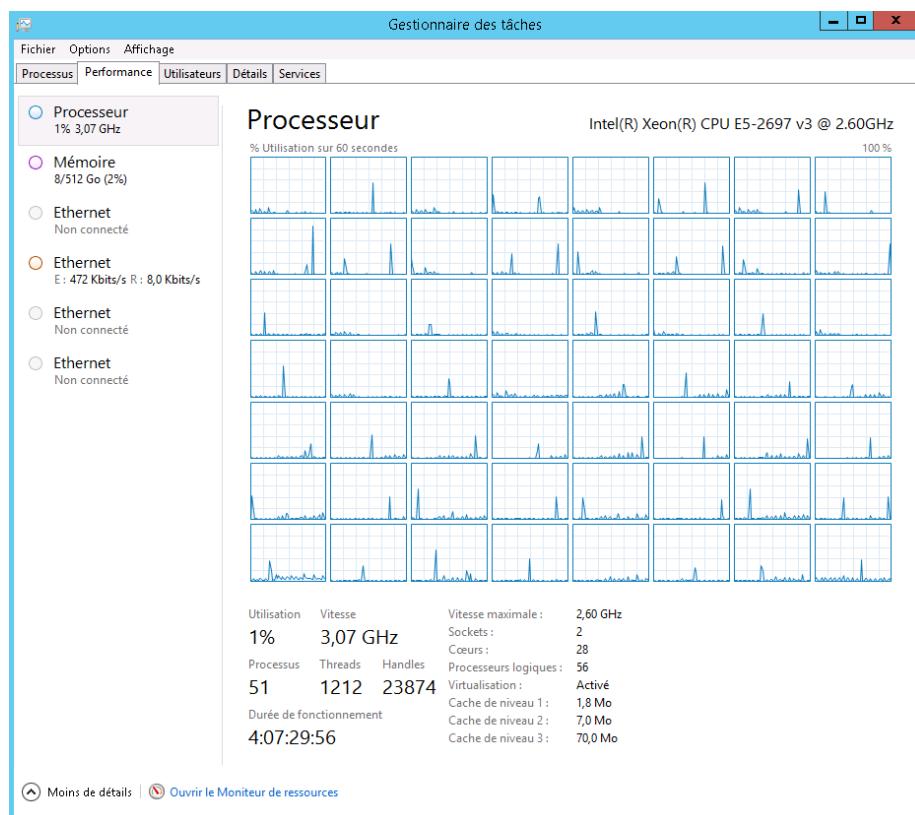


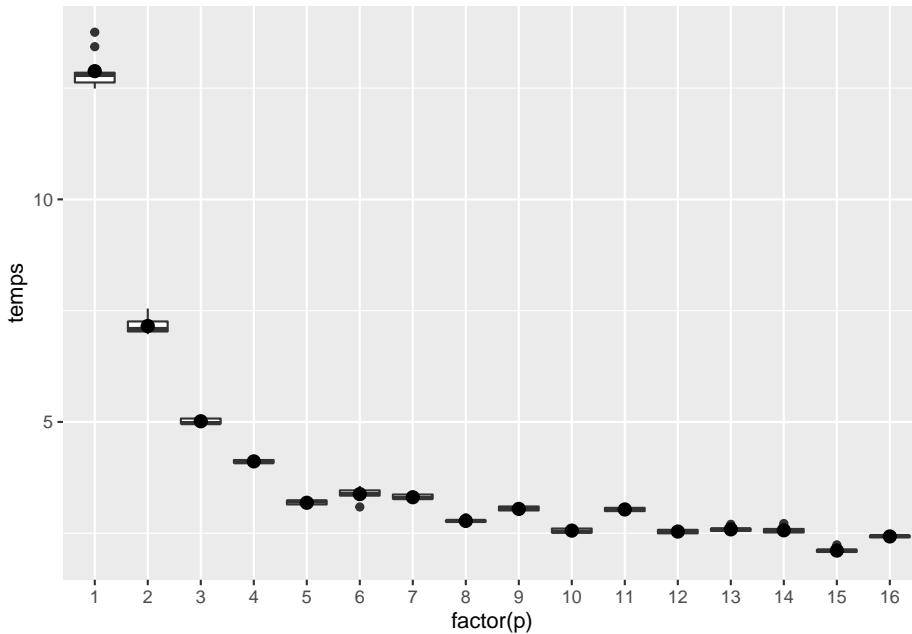
Figure 4.6: Gestionnaire des tâches Windows

4.4.3.3 Choix du nombre de coeurs

Ici, nous avons utilisé 4 coeurs sur les 40 Threads disponibles de la machine, ce qui laisse un nombre considérable de coeurs libres. Sur une machine de type perso, il est d'usage de ne pas utiliser tous les coeurs disponibles afin d'en laisser un certain nombre de libres pour rendre le fonctionnement de la machine stable (qui doit en effet gérer d'autres processus que **R**). Sur un serveur dédié au calcul, il est a priori possible d'utiliser tous les coeurs disponibles à cet usage.

Le gain en temps de calcul n'est pas linéaire en fonction du nombre de coeurs utilisés. Cependant, dans l'exemple ci-dessus, en utilisant 4 coeurs plutôt qu'un seul, on a eu un facteur environ 2.7 de gain en temps calcul. On s'est amusé à refaire le même calcul que précédemment, mais en faisant évoluer le nombre de coeurs alloué. Pour chaque cœur, on a répliqué 10 fois le calcul pour avoir une distribution du temps de calcul. Dans la figure ci-dessous, on a représenté la courbe du temps de calcul en fonction du nombre de coeurs alloués. On constate que :

- la tendance est décroissante mais pas linéaire
- à partir de 5 coeurs, le temps gagné est de moins en moins significatif et on a une asymptote à partir de 10 coeurs.



La raison est qu'en faisant du calcul parallèle, il y a des flux d'informations qui communiquent entre le programme maître et les coeurs sollicités et que ces flux coûtent du temps. Aussi, parfois il arrive que les flux d'informations coûtent plus en temps que les calculs à proprement dit. Autrement dit, il peut arriver que le calcul parallèle ait un effet négatif sur le temps... Nous verrons un exemple

plus tard.

4.4.4 Récupérer les résultats

En utilisant la fonction `sapply()` dans le cas non parallèle, le résultat est stocké sous forme d'un **array**. Avec la fonction `clusterApply()`, le résultat est stocké sous forme de **list**. Dans les exemples précédents, une façon de récupérer les résultats est donc la suivante :

```
resultats$res_par <- unlist(res_par)
```

On calcule ensuite la moyenne et l'écart-type en fonction des valeurs prises par **r_values**:

```
aggregate(resultats[, 2:3], list(r_values = as.factor(resultats[, "r_values"])),
          function(x) c(MEAN = mean(x), SD = sd(x)))

##   r_values res_non_par.MEAN res_non_par.SD res_par.MEAN res_par.SD
## 1      10     4.155421997   2.472656827  5.345868959  3.337169270
## 2     1000     4.916073620   0.314449683  4.933190842  0.283616142
## 3    1e+05     5.004018285   0.035245747  5.001317638  0.028541400
## 4    1e+07     5.000064298   0.003280121  5.001150306  0.003045339
```

On peut également utiliser la syntaxe **tidyverse** :

```
require("dplyr")
resultats %>%
  group_by(r_values) %>%
  summarise_all(list(mean = mean, sd = sd))

## # A tibble: 4 x 5
##   r_values res_non_par_mean res_par_mean res_non_par_sd res_par_sd
##   <dbl>        <dbl>        <dbl>        <dbl>        <dbl>
## 1 10          4.16         5.35         2.47         3.34
## 2 1000        4.92         4.93         0.314        0.284
## 3 1e+05        5.00         5.00         0.0352       0.0285
## 4 1e+07        5.00         5.00         0.00328      0.00305
```

Nous verrons un peu plus tard un équivalent de `sapply()` en calcul parallèle qui permettra de récupérer les résultats sous forme simplifiée.

4.4.5 Utiliser des packages, objets, jeux de données sur les différents coeurs

Lorsqu'on lance un calcul en parallèle sur 4 coeurs, c'est comme si on ouvrait 4 nouvelles consoles **R**. Or, à l'ouverture d'une nouvelle console **R**, il n'y a par défaut aucun package ni objets chargés. C'est pourquoi si le programme fait appel à des librairies ou des objets, l'utilisateur devra le spécifier.

4.4.5.1 Utiliser des packages sur plusieurs coeurs

On reprend la fonction précédente dans laquelle on aimerait changer la loi de distribution gaussienne par une loi de Pareto. La fonction `rpareto()` du package **VGAM** permet de faire cela. On a plusieurs possibilités pour programmer la fonction.

Solution 1 : elle consiste à utiliser la fonction `library()` à l'intérieur de la fonction `myfun_pareto()`. Dans ce cas, si on lance un calcul en parallèle, la librairie sera chargée dans chaque cœur appelé.

```
myfun_pareto <- function(r, scale = 1, shape = 10) {
  library("VGAM")
  mean(rpareto(r, scale = scale, shape = shape))
}
```

Solution 2 : elle consiste à utiliser la syntaxe suivante qui évite de charger toutes les fonctions du package **VGAM** mais qui indique dans quelle librairie il faut aller chercher la fonction `rpareto()`.

```
myfun_pareto <- function(r, scale = 1, shape = 10) {
  mean(VGAM::rpareto(r, scale = scale, shape = shape))
}
```

Dans les deux cas, lorsqu'on executera cette fonction en parallèle, chaque cœur saura où trouver la fonction `rpareto()`. Pour exécuter la fonction en parallèle :

```
r_values <- rep(c(10, 1000, 100000), each = 25)
cl <- makeCluster(P)
res_par <- clusterApply(cl, r_values, fun = myfun_pareto,
                        scale = 1, shape = 10)
stopCluster(cl)
```

Solution 3 : une façon alternative de procéder est d'écrire la fonction sans faire appel à la librairie **VGAM**.

```
myfun_pareto <- function(r, scale = 1, shape = 10) {
  mean(rpareto(r, scale = scale, shape = shape))
}
```

En revanche, il faudra indiquer dans le programme maître, qu'on souhaite charger le package **VGAM** sur tous les coeurs que nous allons utiliser. Ceci se fait à l'aide de la fonction `clusterEvalQ()`. Voici un exemple d'utilisation :

```
cl <- makeCluster(P)

clusterEvalQ(cl, library("VGAM"))
res_par <- clusterApply(cl, r_values, fun = myfun_pareto,
                        scale = 1, shape = 10)
```

```
stopCluster(cl)
```

4.4.5.2 Charger des objets, fonctions ou jeux de données dans les différents coeurs

On reprend l'exemple précédent dans lequel on modifie légèrement la fonction de telle sorte que les paramètres **scale** et **shape** ne sont pas reconnus en tant que variables locales.

```
myfun_pareto <- function(r) {
  mean(rpareto(r, scale = scale, shape = shape))
}
```

Pour que la fonction ne retourne pas de messages d'erreurs, il faudra donc que les objets **scale** et **shape** soient définis en tant que variables globales, et ceci dans chaque cœur. Il est possible de faire cela, toujours grâce à la fonction *clusterEvalQ()* que nous avons utilisée précédemment. Dans l'exemple suivant, la librairie **VGAM** sera chargée dans chaque cœur et les objets **scale** et **shape** seront également définis.

```
cl <- makeCluster(P)

clusterEvalQ(cl, {
  library("VGAM")
  scale <- 1
  shape <- 10
})
res_par <- clusterApply(cl, r_values, fun = myfun_pareto)

stopCluster(cl)
```

Une autre façon est de définir ces objets depuis la session maître :

```
scale <- 1
shape <- 10
```

puis d'exporter ces objets vers tous les coeurs qui seront utilisés dans la suite à l'aide de la fonction *clusterExport()* :

```
cl <- makeCluster(P)

clusterExport(cl, c("scale", "shape"))
clusterEvalQ(cl, library("VGAM"))
res_par <- clusterApply(cl, r_values, fun = myfun_pareto)

stopCluster(cl)
```

Cette méthode peut s'avérer intéressante pour exporter des jeux de données vers

les différents coeurs.

4.4.6 Fonctions *lapply()*, *sapply()*, *apply()*, *mapply()*

Il existe des versions parallélisées de ces fonctions. Celles-ci sont nommées :

- *parLapply()*,
- *parSapply()*,
- *parApply()*,
- *clusterMap()*.

Ces fonctions font appel à la fonction *clusterApply()*. Elles semblent en général un peu plus longue en temps de calcul mais permettent de simplifier la syntaxe de sortie (fonction *parSapply()*) ou d'utiliser des arguments différents en fonction de l'itération (fonction *clusterMap()*).

Exemple : dans le premier exemple de ce chapitre, nous avons utilisé la fonction *sapply()* sur la fonction *myfun()* ainsi

```
res_non_par <- sapply(r_values, FUN = myfun,
                      mean = 5, sd = 10) # options de la fonction myfun
```

Pour la version parallèle, on aurait pu remplacer simplement *sapply()* par *parSapply()* :

```
P <- 4 # définir le nombre de coeurs
cl <- makeCluster(P) # réserve 4 coeurs - début du calcul
system.time(
  res_par <- parSapply(cl, r_values, FUN = myfun, # évalue myfunc() sur r_values
                        mean = 5, sd = 10)
)

## utilisateur      système     écoulé
##       0.003        0.000    16.030
stopCluster(cl) # libère 4 coeurs - fin du calcul
```

Remarque : dans cet exemple, le gain en temps de calcul n'est pas aussi prononcé que lorsqu'on avait utilisé la fonction *clusterApply()* seul. Ceci peut s'expliquer par le fait qu'il y a des opérations supplémentaires avec la fonction *parSapply()*. Aussi, même si l'avantage ici est que le résultat est retourné sous forme de vecteur, on recommande d'utiliser la fonction *clusterApply()*.

4.4.7 Autres packages de calcul parallèle

- **snowFT** : ce package permet de gérer le choix des graines de simulation de façon optimale à l'intérieur de chaque cœur. Nous en présenterons un exemple à la fin de ce chapitre car son utilisation semble prometteuse.

- **foreach** : ce package permet de faire des boucles de type **for** en utilisant une syntaxe similaire. Le but est de faire tourner en parallèle les instructions à l'intérieur de la boucle **for**. Ce package est en général couplé avec le package **doParallel**, via le package **parallel** (voir vignette à ce lien), dont on présente ci-dessous un exemple d'utilisation :

```
require("doParallel")
getDoParWorkers() # affiche nombre de coeurs alloué

## [1] 1

registerDoParallel(cores = P) # alloue le nombre de coeurs souhaité
getDoParWorkers() # Pour vérifier qu'on utilise bien tous les coeurs

## [1] 4

system.time(
  res_par_foreach <- foreach(i = r_values)
    %dopar% myfun(i, mean = 5, sd = 10)
  )

## utilisateur      système     écoulé
##       11.841        0.736     5.086
# pour revenir au nombre de cœur initial
registerDoParallel(cores = 1)

# présentation des résultats
resultats$res_par_foreach <- unlist(res_par_foreach)
```

- **doMPI** : utilise une architecture mpi.

Des temps de calcul ont été comparés entre ces différentes solutions (voir Introduction to parallel computing with R), le package **parallel** est parmi ceux qui obtiennent les meilleurs résultats.

Exercice 3.2.

Le bagging est une technique utilisée pour améliorer la classification notamment celle des arbres de décision. On va programmer l'algorithme suivant :

Entrées :

- **ech_test** l'échantillon test,
- **ech_appr** l'échantillon d'apprentissage,
- **B** le nombre d'arbres,

Pour $k = 1, \dots, B$:

1. Tirer un échantillon bootstrap dans **ech_appr**
2. Construire un arbre CART sur cet échantillon bootstrap et prédire sur l'échantillon test.

On va appliquer cet algorithme sur le jeu de données **iris** qui est inclus dans **R** par défaut. L'objectif est de prédire à quel type d'espèce appartient une fleur (variable **Species** qui contient 3 variétés) en fonction de ses caractéristiques (variables **Sepal.Length**, **Sepal.Width**, **Petal.Length** et **Petal.Width**).

Tout d'abord, on définit l'échantillon **ech_test** qui contient les observations à prédire. Ici, on en tire 25 au hasard et les 125 observations restantes constitueront l'échantillon d'apprentissage :

```
set.seed(1)
id_pred <- sample(1:150, 25, replace = F)
ech_test <- iris[id_pred, ]
ech_appr <- iris[-id_pred, ]
```

- Créer la fonction *class_tree()* qui prend comme argument d'entrée la valeur de la graine *k* utilisée pour tirer un échantillon bootstrap de **ech_appr** (il s'agit simplement d'un tirage aléatoire avec remise appliquée après la fonction *set.seed(k)*), va construire un arbre CART sur cet échantillon bootstrap et retournera la prédiction sur l'échantillon test **ech_test**. On pourra utiliser les fonctions *rpart()* et *predict.rpart()*, mais l'objet retourné sera un vecteur de **character** contenant l'espèce prédite.

Le résultat de cette fonction est le suivant :

```
require("rpart")
class_tree(1)

## [1] "versicolor" "virginica"   "setosa"      "setosa"      "versicolor"
## [6] "versicolor" "setosa"       "virginica"   "versicolor"   "setosa"
## [11] "versicolor" "versicolor"   "setosa"      "virginica"   "virginica"
## [16] "setosa"     "virginica"   "virginica"   "versicolor"   "setosa"
## [21] "virginica"  "versicolor"  "virginica"   "setosa"      "setosa"
```

- A présent, nous allons répéter 100 fois cette opération en effectuant du calcul parallèle. Pour cela, on aura besoin d'exporter dans les différents coeurs la librairie **rpart** et les objets suivants **ech_test**, **ech_appr**:
- Récupérer les données et donner les valeurs prédites pour chaque observation de l'échantillon test.
- Calculer le tableau de bien classés

4.5 Equilibrer la répartition des tâches

En envoyant plusieurs tâches dans différents coeurs, il se peut que certains coeurs soient plus sollicités que d'autres. On considère la fonction suivante qui consiste à calculer la moyenne d'un échantillon de taille **r** simulée selon une loi gaussienne.

```
rnmean <- function(r, mean = 0, sd = 1) {
  mean(rnorm(r, mean = mean, sd = sd))
}
```

Nous allons appliquer cette fonction en utilisant des valeurs de **r** qui soient très hétérogènes de telle sorte qu'on va créer un déséquilibre dans l'exécution des tâches.

```
N <- 40
set.seed(50)
r.seq <- sample(ceiling(exp(seq(7, 14, length = 50))), N)
r.seq

## [1] 903730    4576   79676 1202605  679133    1460    2981    9348   12439
## [10]  44995   19095  332464   16553   187749  122307   2585   51904   59875
## [19]  39005   3967   33813    6090    1266   69069  162755   8104  783424
## [28] 1042512 141089  288206    5279   383519   1684   2241  22027  510353
## [37]  10783   1942  249839  106025
```

Si on parallélise sur 4 coeurs, comment vont se répartir l'envoi des tâches sur les coeurs ? Cela va se faire automatiquement de la façon suivante :

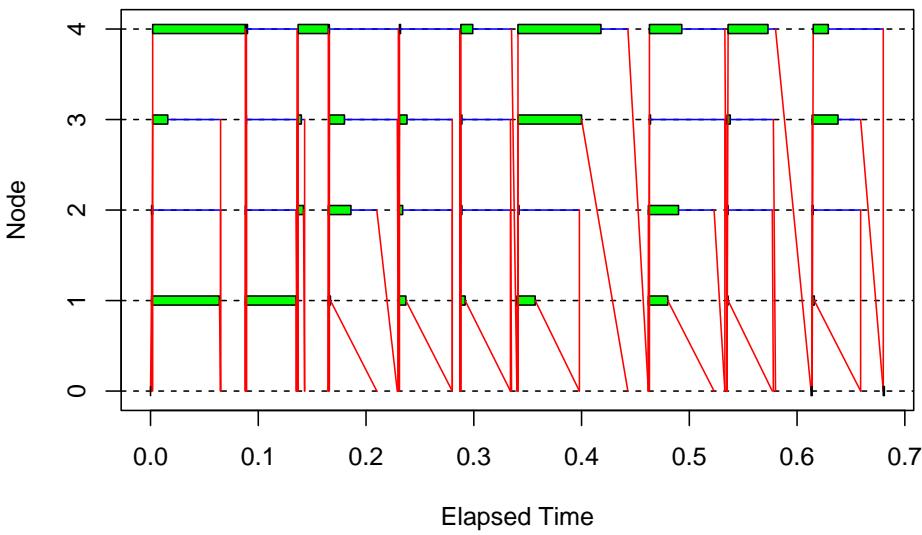
- le cœur 1 va effectuer le calcul pour les valeurs de **r** suivantes : 162755 (1ère position de **r.seq**), 29311 (5ème position de **r.seq**), 1266 (9ème position de **r.seq**), etc.
- le cœur 2 va effectuer le calcul pour les valeurs de **r** suivantes : 22027 (2ème position de **r.seq**), 1460 (6ème position de **r.seq**), 1942 (10ème position de **r.seq**), etc.
- le cœur 3 va effectuer le calcul pour les valeurs de **r** suivantes : 3967 (3ème position de **r.seq**), 79676 (7ème position de **r.seq**), 9348 (11ème position de **r.seq**), etc.
- le cœur 4 va effectuer le calcul pour les valeurs de **r** suivantes : 187749 (4ème position de **r.seq**), 51904 (8ème position de **r.seq**), 4576 (12ème position de **r.seq**), etc.

On constate que le 1er cœur va d'abord commencer à simuler un vecteur de taille 162755 alors que le second cœur va simuler un vecteur de taille 22027. Le second cœur devrait donc être monopolisé moins de temps que le premier. On peut alors se poser la question s'il va passer au calcul de sa seconde valeur à calculer une fois le premier calcul terminé.

Il est possible de faire un rapport de l'usage des coeurs grâce à la fonction *snow.time()* du package **snow**.

```
library("snow")
cl <- makeCluster(4)
ctime1 <- snow.time(clusterApply(cl, r.seq, fun = rnmean))
plot(ctime1, title = "Usage with clusterApply")
```

Usage with `clusterApply`

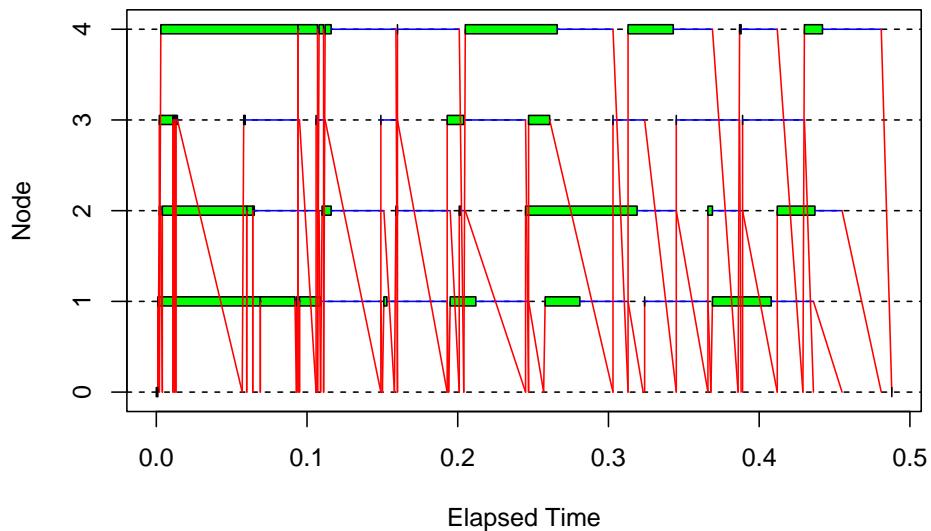


Dans le graphique ci-dessus, les traits en vert correspondent à des périodes où un coeur i ($i = 1, \dots, 4$ en ordonnée) est en train d'effectuer un calcul. Un trait bleu correspond à une période où le coeur est en repos. Aussi, on constate que dans un coeur donné, pour passer à l'exécution d'une nouvelle tâche, il faut attendre que toutes les instructions effectuées sur les coeurs en parallèle soient terminées. Autrement dit, le coeur 2 a du attendre que le coeur 1 ait terminé l'exécution de sa première tâche avant de pouvoir passer à la tâche suivante. Ceci n'est donc pas optimale.

Une alternative à la fonction `clusterApply()` est d'utiliser la fonction `clusterApplyLB()` qui a été optimisée pour cet usage. On constate ci-dessous que toutes les tâches n'ont plus besoin d'avoir été exécutées avant de passer aux suivantes.

```
cl <- makeCluster(P)
ctimeLB <- snow.time(clusterApplyLB(cl, r.seq, fun = rnmean))
plot(ctimeLB, title = "Usage with clusterApplyLB")
```

Usage with clusterApplyLB



```
stopCluster(cl)
```

Le gain en temps de calcul est d'un facteur 1.4.

Remarque : dès lors que nous avons chargé la librairie **snow**, ce sont les fonctions *clusterApply()* et *clusterApplyLB()* du package **snow** qui ont été utilisées alors qu'elles existent simultanément dans les packages **snow** et **parallel**. Elles sont quasiment équivalentes d'un package à un autre, mais pour garder en mémoire le rapport sur l'usage des coeurs, il faut utiliser les fonctions du package **snow**.

4.6 Améliorer la répartition des tâches

Dans l'exemple précédent, on a vu qu'il y avait sans arrêt des flux d'information entre le programme maître et les coeurs car on a 40 calculs ou ‘jobs’ qui sont envoyés au fur et à mesure dans les coeurs alloués. Une façon de déjouer cela est de re-travailler le programme maître pour indiquer que les 10 premières tâches seront envoyées dans le 1er coeur, les 10 suivantes dans le second, etc. Pour cela, on modifie d'abord la fonction *rnmean()* afin qu'elle puisse s'appliquer avec **r** défini comme un vecteur, plutôt qu'un scalaire :

```
rnmean_vect <- function(r, mean = 0, sd = 1) {
  sapply(r,
         function(x) mean(rnorm(x, mean = mean, sd = sd)))
}
```

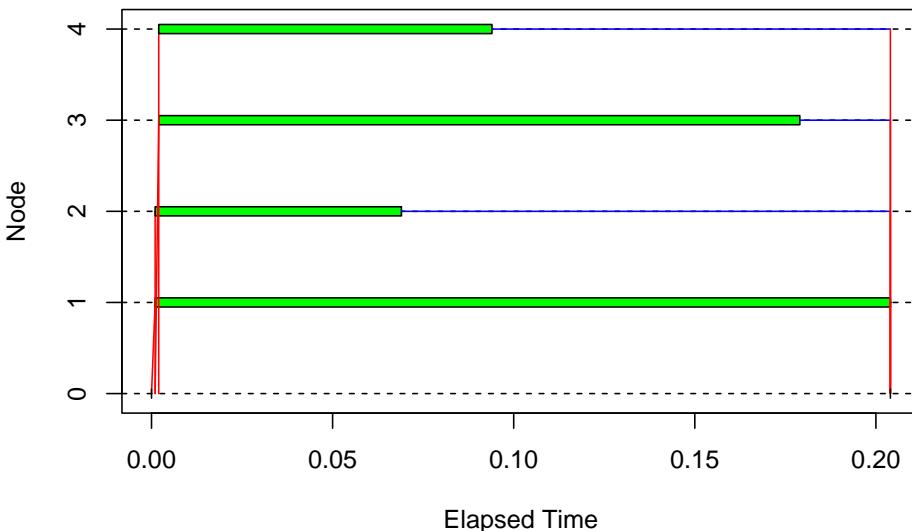
Ensuite, on transforme le vecteur **r.seq** en liste composée de 4 sous vecteurs.

```
r.seq_list <- list(r.seq[1:10],
                     r.seq[11:20],
                     r.seq[21:30],
                     r.seq[31:40])
```

Enfin, on refait appel à la fonction

```
library("snow")
cl <- makeCluster(P)
ctime <- snow.time(clusterApply(cl, r.seq_list, fun = rnmean_vect))
plot(ctime, title = "Usage with clusterApply")
```

Usage with clusterApply



```
stopCluster(cl)
```

A travers cet exemple, on voit que les flux d'informations sont minimes et le temps de calcul par conséquent meilleur avec une amélioration d'un facteur 3.3.

Exercice 3.3.

En vous inspirant de cette section, améliorer la fonction `class_tree()` vue précédemment en la vectorisant. Le but est de ne faire que 4 jobs (25 bootstrap par job) au lieu de 100. Comparer le temps de calcul avec la version non vectorisée.

4.7 Fonction vectorisée VS calcul // VS code C++

Faire du calcul // n'est pas nécessairement bénéfique si celui-ci n'est pas utilisé dans les règles de l'art. Dans cette section, on va comparer plusieurs façons de coder le même problème.

On considère le jeu de données suivant qui prend un peu moins d'1 Go de mémoire vive (10M d'observations et 3 variables).

```
n <- 10000000
big_file <- data.frame(chiffre = 1:n,
                       lettre = paste0("caract", 1:n),
                       date = sample(seq.Date(as.Date("2017-10-01"),
                                              by = "day", len = 100), n,
                                     replace = T))
object.size(big_file)

## 840001520 bytes
```

L'objectif est de créer une nouvelle variable binaire qui vaut 1 si la variable **chiffre** est paire est 0 sinon. Pour cela, on va comparer plusieurs moyens pour y arriver.

Solution 1 : on va utiliser la fonction *ifelse()* qui s'applique sur la fonction **%%**.

```
sol_1 <- microbenchmark::microbenchmark({
  big_file$new <- ifelse(big_file$chiffre %% 2 == 0, 1, 0)
}, times = 10L
)
```

Solution 2 : on va utiliser les opérateurs d'affectation et la fonction **%%**

```
sol_2 <- microbenchmark::microbenchmark({
  big_file$new <- as.numeric(big_file$chiffre %% 2 == 0)
}, times = 10L
)
```

Solution 3 : on va utiliser du calcul // avec la fonction *foreach()*. D'abord, on crée la fonction à paralléliser qui regarde si un chiffre est pair ou non :

```
compare <- function(x)
  x %% 2 == 0
```

Ensuite, on parallélise avec la fonction *foreach()* (ici, sur les 1000 premières valeurs uniquement car le temps de calcul serait trop long sur l'ensemble des individus) :

```
require("doParallel")
P <- 4
registerDoParallel(cores = P)
system.time(
  res <- foreach(i = 1:1000) %dopar%
    compare(big_file$chiffre[i])
)
```

Dans cet exemple, on a mal programmé la parallélisation comme cela a été vue dans la section 1.6. On va donc re-programmer la fonction à paralléliser pour indiquer que l'on souhaite faire le calcul des 2500000 premières observations sur le 1er cœur, les 2500000 suivantes sur le second cœur, etc. Comme la fonction *compare()* est déjà vectorisée, ce n'est pas la peine de la changer. En revanche, on change l'appel de la fonction *foreach()* afin de l'adapter à ce que l'on souhaite faire :

```
require("doParallel")
sol_3 <- microbenchmark::microbenchmark({
  registerDoParallel(cores = P)
  res <- foreach(i = 1:4) %dopar%
    compare(big_file[(1 + 2500000 * (i - 1)):(2500000 * i)],
  }, times = 10L
})
```

Solution 4 : on va utiliser du code C++

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector compare_cpp(IntegerVector x) {
  int n = x.size();
  IntegerVector res(n);

  for(int i = 0; i < n; i++) {
    if(x(i) % 2 == 0) {
      res(i) = 1;
    } else {
      res(i) = 0;
    }
  }

  return res;
}

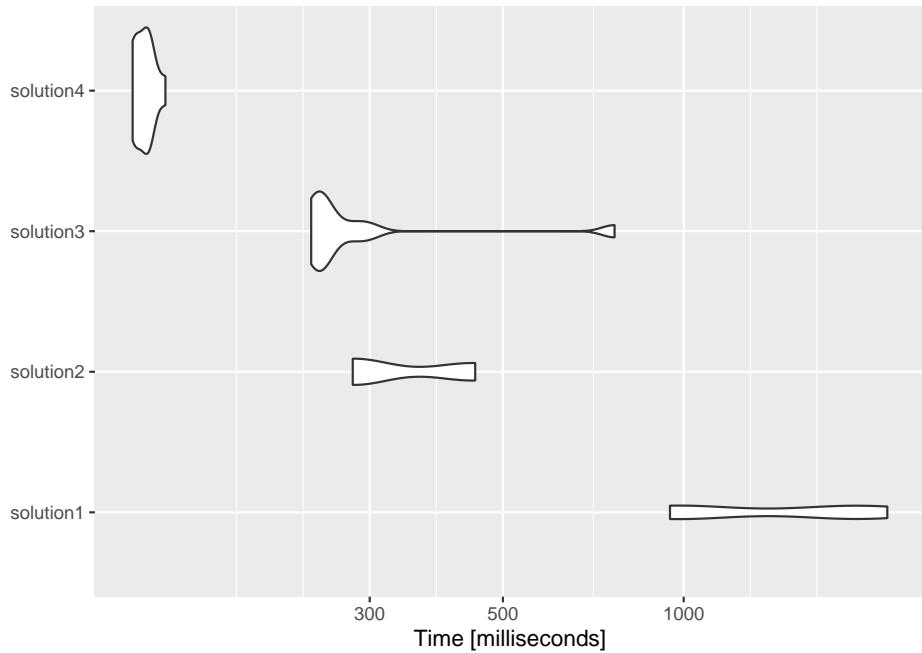
require("Rcpp")
sol_4 <- microbenchmark::microbenchmark(
```

```
big_file$new <- compare_cpp(big_file$chiffre),
times = 10L)
```

Représentons les performances de ces 4 solutions :

```
time_mbm <- rbind(sol_1, sol_2, sol_3, sol_4)
time_mbm$expr <- paste0("solution", rep(1:4, each = 10))
ggplot2::autoplot(time_mbm)
```

```
## Coordinate system already present. Adding new coordinate system, which will replace
```



Si on compare les 4 solutions, c'est celle qui utilise C++ qui est la plus performante. Souvent, lorsqu'un programme compte un grand nombre de boucles, c'est effectivement cette solution qui est la meilleure. Ici, le calcul // (à condition que la méthode soit bien implantée) donne des résultats équivalents à la solution 2 qui utilise la commande %. Il n'y a pas d'améliorations à utiliser du code // ici car les flux d'informations entre le programme maître et les coeurs sont importants (en effet, on transfère une grosse quantité de données). Enfin, on constate que la solution 1 prend quant à elle plus de temps que la solution 2 car la fonction *ifelse()* contient pas mal de codes internes.

4.8 Reproductibilité des résultats : choix de la graine aléatoire

Il est de plus en plus souvent demander aux programmeurs de coder de telle sorte que leur résultats soient reproductibles par d'autres sur n'importe quelle machine. Lorsqu'on fait des simulations, il est possible de fixer une graine avec la fonction `set.seed()`, mais ceci n'est valable que sur la machine "maître". Ainsi, on peut exécuter autant de fois que l'on souhaite l'instruction suivante, cela donnera des résultats différents à chaque fois car des tirages différents ont été réalisés à chaque itération.

```
cl <- makeCluster(P)
res_par <- parSapply(cl, 1:100, function(x) mean(rnorm(100)))
stopCluster(cl)
```

Une façon de régler ce problème est de fixer une graine à l'intérieur de la fonction qu'on parallélise (comme ce qui a déjà été fait précédemment) :

```
rnmean <- function(x, r, mean = 0, sd = 1) {
  set.seed(x)
  mean(rnorm(r, mean = mean, sd = sd))
}

cl <- makeCluster(P)
res_par <- parSapply(cl, 1:100, rnmean,
                      r = 100, mean = 0, sd = 1)
stopCluster(cl)
```

Une autre façon de faire est d'utiliser la fonction `performParallel()` du package `snowFT` qui gère parfaitement la gestion des graines aléatoires et fait en sorte d'attribuer dans chaque coeur des graines qui pourront être reproduites.

```
rnmean <- function(r, mean = 0, sd = 1) {
  mean(rnorm(r, mean = mean, sd = sd))
}

library("snowFT")
seed <- 1
r_values <- rep(c(10, 1000, 100000, 10000000), each = 10)
res <- performParallel(P, r.seq, fun = rnmean,
                        seed = seed)
tail(unlist(res))

## [1] 0.0020074546 0.0001398813 -0.0028261112 0.0277917239 0.0001672218
## [6] -0.0024037920
```

Remarque : cette fonction permet également de définir ou exporter des objets/librairies vers les différents coeurs en utilisant les options `initexpr` et `ex-`

port. En reprenant l'exemple précédent avec la fonction *myfun_pareto()* :

```
myfun_pareto <- function(r) {
  mean(rpareto(r, scale = scale, shape = shape))
}

seed <- 1
scale <- 1
shape <- 10
r_values <- rep(c(10, 1000, 100000, 10000000), each = 10)

res <- performParallel(P, r.seq, fun = myfun_pareto,
                        seed = seed,
                        initexpr = require("VGAM"),
                        export = c("scale", "shape"))
tail(unlist(res))

## [1] 1.111299 1.111008 1.110563 1.110589 1.111495 1.111505
```

Chapter 5

Visualisation de données

Packages à installer

Depuis le CRAN :

```
install.packages(c("devtools",      # devtools
                   "dplyr",          # piping
                   "gapminder",       # données gapminder
                   "ggcorrplot",     # graphiques corrélation
                   "ggridges",        # données ggridges
                   "ggplot2",         # graphiques ggplot2
                   "ggpol",           # jitter et boxplot
                   "kableExtra",      # tableaux customisés
                   "RColorBrewer",    # couleurs
                   "stargazer",       # tableaux
                   "survival",         # modèle de survie
                   "survminer",        # plot de survie
                   "plotly",           # graphiques interactifs
                   "vcd",              # mosaic plot
                   "visreg"),          # effets conditionnels des modèles
               dependencies = TRUE)
```

5.1 Fonctions graphiques de base VS ggplot2

Il est possible de réaliser des graphiques statistiques sous **R** de multiples façons. Même si de nombreuses fonctions provenant de différents packages font la même chose, deux stratégies semblent se dégager pour réaliser des graphiques statistiques :

- utiliser les fonctions de base *plot()*, *hist()*, *barplot()*, *boxplot()*, etc. conjointement avec les fonctions de bas-niveau *lines()*, *text()*, *legend()* pour “orne-

menter'' le graphique et la fonction `par()` pour modifier les paramètres graphiques (marges, taille de la fenêtre graphique, taille des légendes, etc.).

- utiliser la philosophie du package **ggplot2** proposée par H. Wickham.

En général, les utilisateurs décident d'adopter l'une ou l'autre façon de faire, rarement les deux. L'idée de cette section est d'essayer de faire une revue des avantages de l'un et de l'autre et si possible déterminer à quel moment il est intéressant d'utiliser l'un plutôt que l'autre.

La stratégie a été la suivante : une grande majorité des utilisateurs de **ggplot2** choisissent d'utiliser ce package parce que celui-ci leur permet de réaliser des graphiques élégants en modifiant le moins possible les paramétrages par défaut. On va donc partir de graphiques obtenus avec **ggplot2**, supposés "élégants" et essayer d'obtenir des graphiques équivalents en utilisant la syntaxe de base.

Cette section est fortement inspirée de cette page :

- <https://flowingdata.com/2016/03/22/comparing-ggplot2-and-r-base-graphics/>

Pour une introduction aux graphiques de base et au package **ggplot2** :

- http://www.thibault.laurent.free.fr/cours/R_intro/chapitre_4.html

5.1.1 Graphiques standards

Par graphique standard, on entend les graphiques de type diagramme en barre, histogramme, nuage de points, boîtes à moustaches parallèles, etc. Dans le contexte où on compare les fonctions graphiques de base à celles du package **ggplot2**, les graphiques standards seront opposés à des graphiques de types "conditionnels". Les graphiques conditionnels correspondent à des graphiques standards qu'on va réaliser conditionnellement à une variable souvent qualitative.

5.1.1.1 Diagramme en barre

On va commencer par un diagramme en barre. On va utiliser les données **diamonds** qui sont incluses dans le package **ggplot2** et qui donnent des informations sur la vente de plus de 50000 diamants. Parmi les variables observées, le prix, le carat, la qualité de la coupe, la couleur, etc.

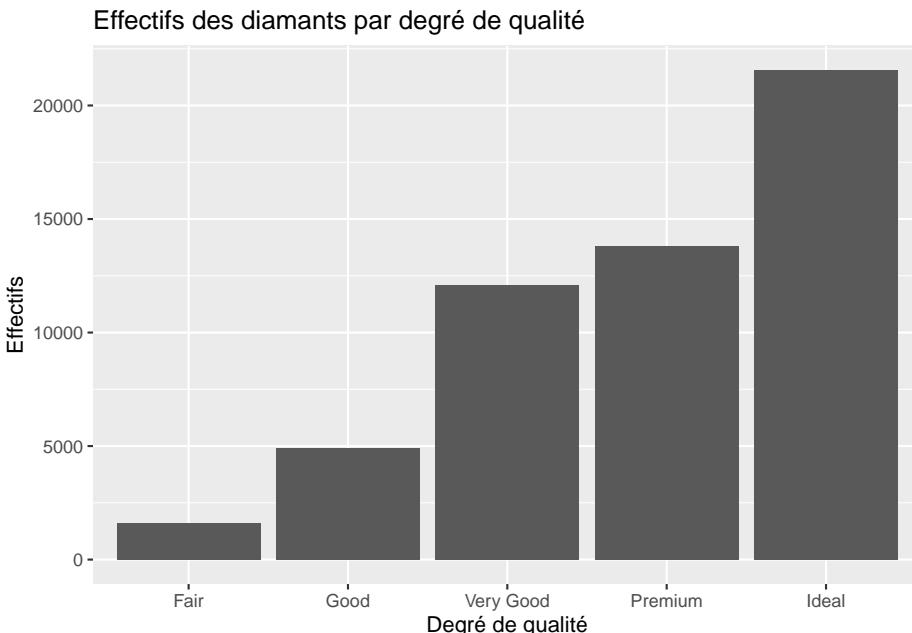
```
require("ggplot2")
data("diamonds")
```

On rappelle qu'avec **ggplot2**, les commandes s'enchaînent avec l'opérateur `+`. La première opération consiste à spécifier le jeu de données sur lequel on travaille (fonction `ggplot()`). Ensuite, on spécifie le nom de la variable qui nous intéresse (avec la fonction `aes()`). Une des spécificités de l'univers **tidyverse** est qu'une

fois le nom du **data.frame** spécifié, il suffit d'appeler directement les variables par leurs noms, sans préciser à quelle **data.frame** elles appartiennent.

Enfin, on donne la fonction correspondant au graphique que l'on souhaite représenter. Pour un diagramme en barre, cela se fait avec la fonction *geom_bar()*. Si on souhaite utiliser des couleurs différentes par barre, une façon de le faire sans avoir à spécifier aucun nom de couleur, consiste à ajouter dans la fonction *aes()* l'option **fill=** suivi du nom d'une variable qualitative que l'on souhaite représenter. Il peut s'agir de la même variable pour laquelle on souhaite tracer des barres, mais il peut aussi s'agir d'une autre variable (dans ce cas, chaque barre sera découpée par étages selon les modalités de la seconde variable).

```
ggplot(diamonds) +
  aes(x = cut) +
  geom_bar(stat = "count") +
  xlab("Degré de qualité") +
  ylab("Effectifs") +
  ggtitle("Effectifs des diamants par degré de qualité")
```



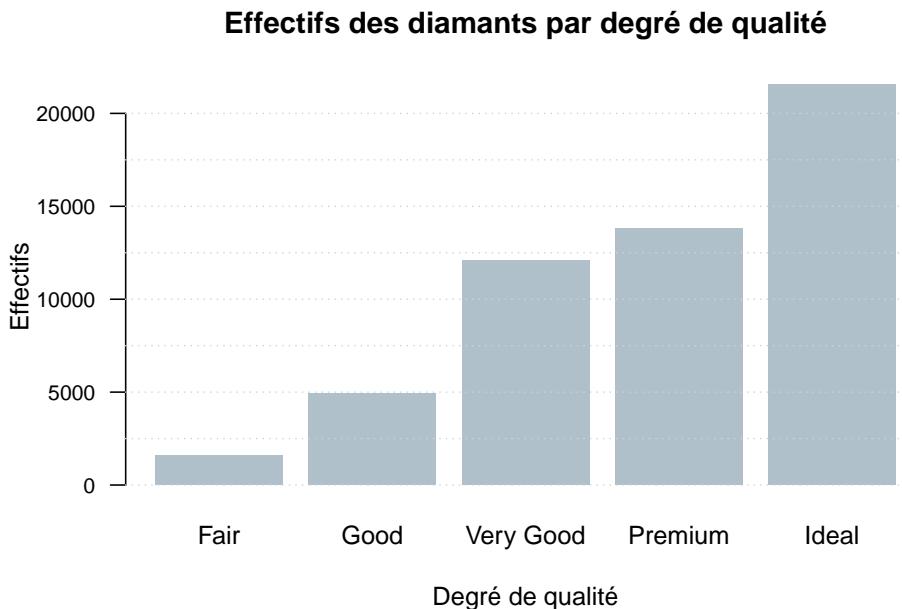
Pour représenter un diagramme en barres avec les graphiques de base, on a besoin de construire une table de contingences associée à une variable qualitative.

```
tab_cut <- table(diamonds$cut)
```

Une des caractéristiques de **ggplot2** est de représenter des lignes verticales et horizontales pour aider à la lecture du graphique. Pour faire cela avec les

graphiques de base, on peut utiliser la fonction `grid()` qui précise le nombre de barres horizontales et verticales à représenter ou bien utiliser la fonction `abline()` qui donne les coordonnées où tracer les droites. Pour représenter la graduation de l'axe des ordonnées horizontalement, on utilise l'option `las=1` dans les paramètres généraux de la fonction `par()`.

```
par(las = 1)
barplot(tab_cut,
        col = "#AFCOCB",
        border = FALSE,
        main = "Effectifs des diamants par degré de qualité",
        xlab = "Degré de qualité",
        ylab = "Effectifs",
        cex.axis = 0.8)
abline(h = seq(0, 20000, by = 2500), col = "lightgray", lty = "dotted")
```



Pour ce type de graphique, il n'y a pas de gros écarts en termes de nombre de lignes de codes entre les deux styles de graphique.

Remarque : en revanche, pour reproduire exactement le même graphique (fond en gris, marges, etc.) en utilisant une syntaxe de base, cela demanderait un peu plus de travail. On le fait ici pour le `barplot` pour que l'utilisateur se rende compte des lignes de codes qui sont implicitement exécutées dans l'appel de fonctions de type `ggplot2` :

```
op <- par(mar = c(3, 3.2, 0.5, 0.5), # marge
          mgp = c(1, 0.5, 0),           # emplacement annot/étiquettes/légendes
          oma = c(0, 0, 1, 0),           # emplacement pour le titre
```

```
cex.axis = 0.75, # tailles des anotations
cex.lab = 0.85, # tailles des légendes
pch = 16,         # type de point utilisé
las = 1,          # affiche les étiquettes /_ aux axes
bty = "n"         # pas de box
)
plot.new()
plot.window(xlim = c(0.5, 5.5), ylim = c(0, max(tab_cut)))

rect(par()$usr[1], par()$usr[3],
      par()$usr[2], par()$usr[4],
      col = "grey89", border = "white")

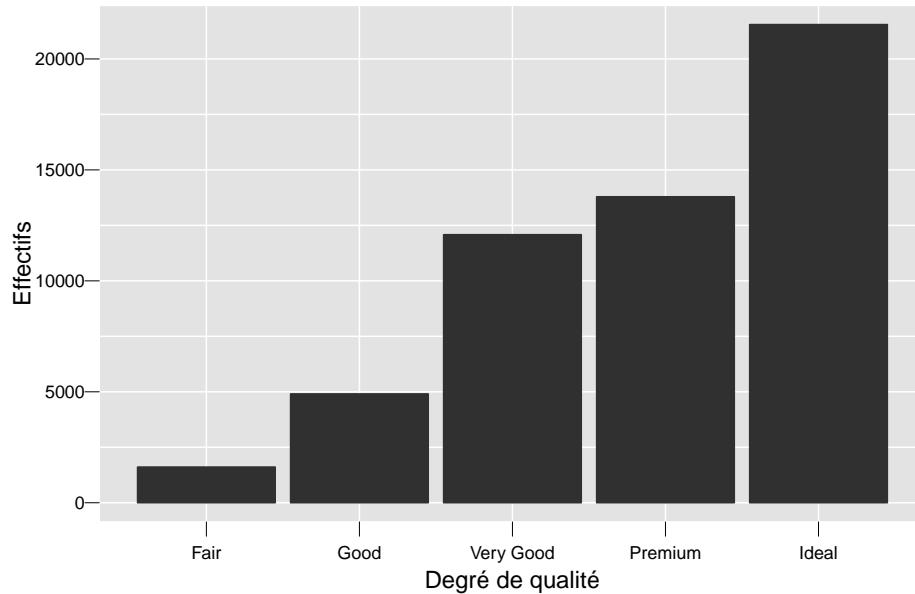
abline(h = seq(0, 25000, 2500), col = "white")
abline(v = seq(1, 5, 1), col = "white")

barre.x <- c(0.55, 1.45)

for (k in 0:4)
  rect(barre.x[1] + k, 0, barre.x[2] + k, tab_cut[k + 1],
       col = "grey19", border = "grey19")

axis(side = 1,
     at = seq(1, 5, 1), # ou mettre les graduations
     labels = c("Fair", "Good", "Very Good", "Premium", "Ideal"), # quelles étiquettes
     lwd = 0,           # supprime la ligne de l'axe des abscisses
     lwd.ticks = 0.25) # donne l'épaisseur des graduations
mtext("Degré de qualité", 1, line = 1.5)
axis(side = 2,
     at = seq(0, 25000, 5000),
     labels = seq(0, 25000, 5000),
     lwd = 0,
     lwd.ticks = 0.25,
     cex = 0.8)
mtext("Effectifs", 2, line = 2.2, las = 3)
mtext("Effectifs des diamants par degré de qualité", 3,
      line = -0.25, las = 1, outer = T, adj = 0)
```

Effectifs des diamants par degré de qualité



5.1.1.2 Séries temporelles

On va considérer une série temporelle mensuelle, supposons qu'il s'agit d'un indice boursier agrégé mensuellement :

```
serie <- c(161.31, 154.00, 161.94, 160.23, 173.20, 170.21, 163.97, 161.70,
         144.91, 145.31, 140.50, 139.58, 135.60, 124.40, 132.24, 150.51,
         146.56, 153.00, 151.78, 160.65, 158.32, 158.06, 153.50, 161.95,
         167.00, 175.00, 180.48, 173.82, 160.05, 152.80, 153.58, 145.00,
         142.98, 145.35)
```

Pour associer des dates à ces valeurs, nous allons utiliser le type d'objet **Date**. On utilise ici une méthode générique de la fonction `seq()` appliquée à des dates :

```
date_serie <- seq(as.Date("2015/1/1"), by = "month", length.out = 34)
```

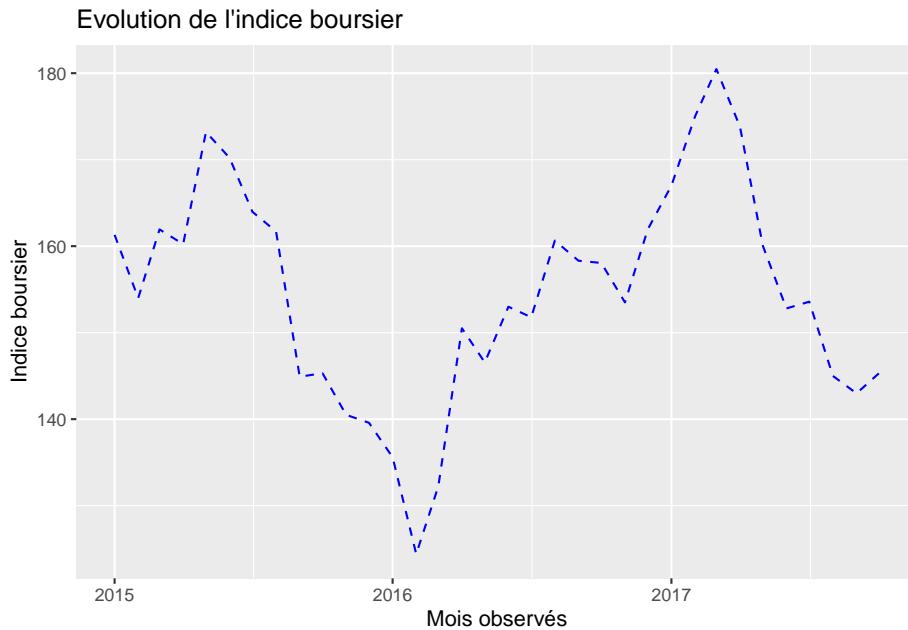
Pour représenter cette série avec **ggplot2**, il faut d'abord créer un **data.frame** :

```
serie_df = data.frame(date_serie = date_serie, serie = serie)
```

Ensuite, il suffit d'utiliser de préciser dans la fonction `aes()` que la variable `date_serie` est associée à `x` et la variable `serie` à `y`. C'est la fonction `geom_line()` qui indique qu'on va relier les points par des lignes brisées :

```
ggplot(serie_df) +
  aes(x = date_serie, y = serie) +
```

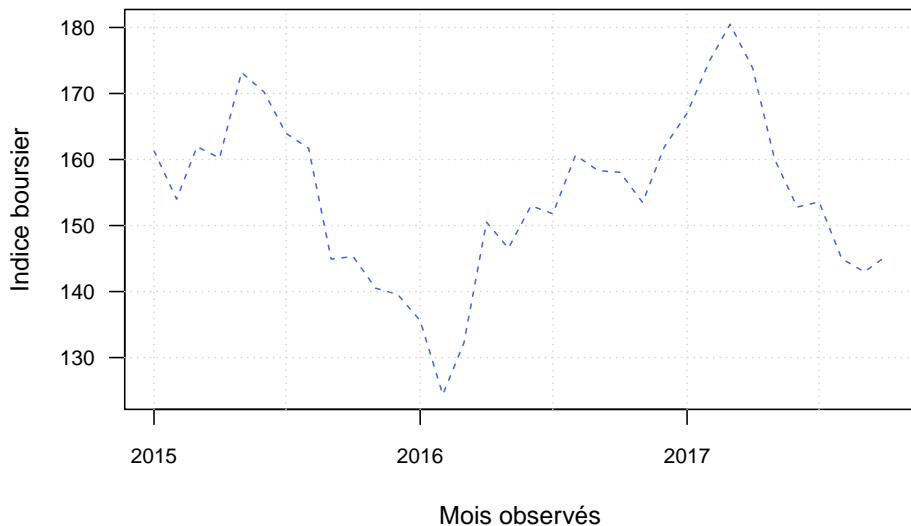
```
geom_line(linetype = 2, colour = "blue") +
xlab("Mois observés") +
ylab("Indice boursier") +
ggtitle("Evolution de l'indice boursier")
```



En utilisant les graphiques de base, cela se fait assez bien car le format **Date** est également reconnu par la fonction générique **plot()**. On obtient donc un résultat similaire avec seulement quelques lignes de code supplémentaire.

```
par(las = 1)
plot(serie ~ date_serie, data = serie_df,
      type = "l",
      col = "royalblue",
      lty = 2,
      main = "Evolution de l'indice boursier",
      xlab = "Mois observés",
      ylab = "Indice boursier",
      cex.axis = 0.8)
abline(h = seq(130, 180, by = 10),
       v = date_serie[seq(1, 32, 6)],
       col = "lightgray", lty = "dotted")
```

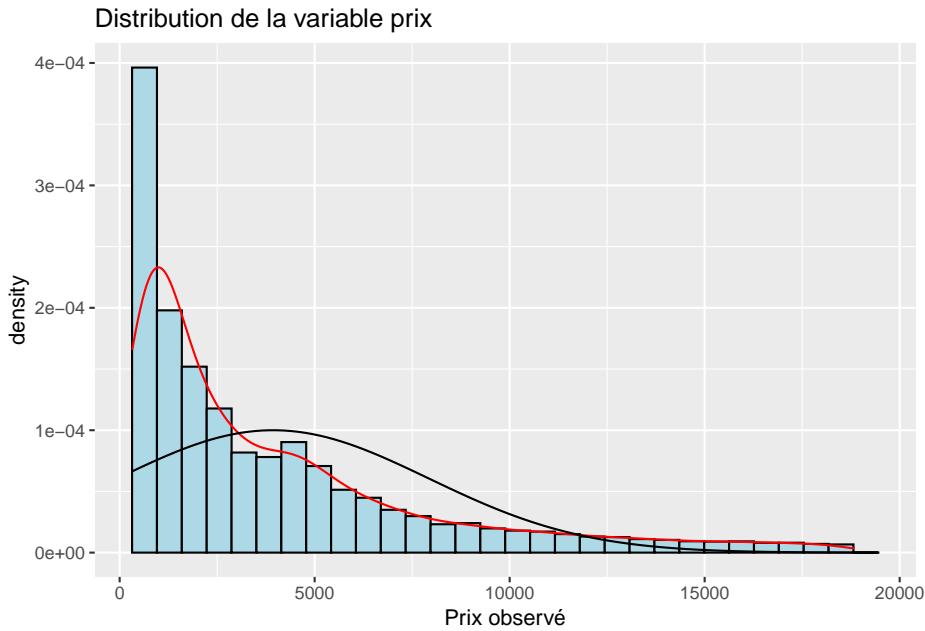
Evolution de l'indice boursier



5.1.1.3 Histogramme et fonction de densité

Pour représenter un histogramme et une densité avec **ggplot2**, cela se fait avec les fonctions *geom_histogram()* et *geom_density()*. L'option **bins=** dans la fonction *geom_histogram()* permet de donner le nombre de barres. L'option **adjust =** permet d'ajuster le degré de lissage de la fonction non paramétrique de la densité. Pour ajouter la fonction paramétrique d'une loi gaussienne, on utilise la fonction *stat_function()* :

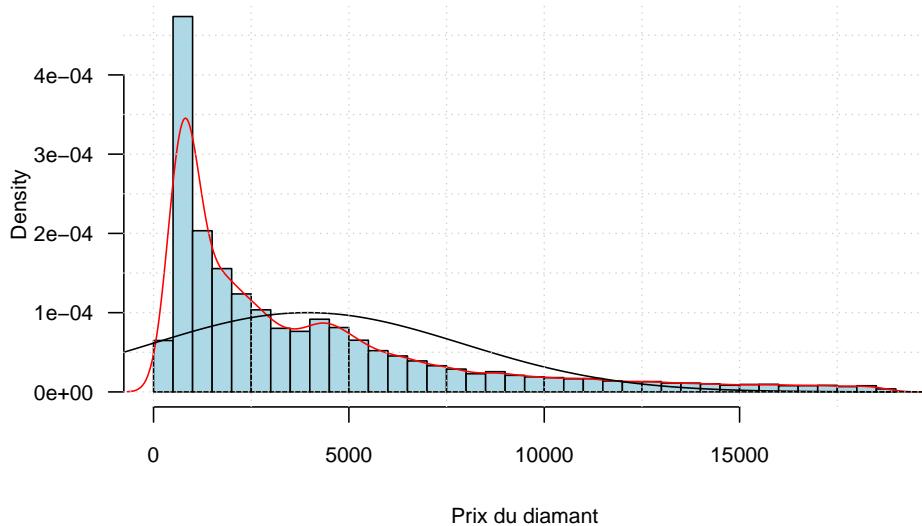
```
ggplot(diamonds) +
  aes(x = price) +
  geom_histogram(aes(y = ..density..), fill = "lightblue", colour = "black",
                 bins = 30) +
  geom_density(colour = "red",
              adjust = 2) +
  stat_function(fun = dnorm,
               args = c(
                 mean = mean(diamonds$price),
                 sd = sd(diamonds$price))) +
  xlab("Prix observé") +
  ggtitle("Distribution de la variable prix")
```



En utilisant les fonctions de base, on peut reproduire quelque chose d'équivalent sans que cela soit trop coûteux. A noter toutefois que l'option **nclass=** de la fonction *hist()* ne donnera pas nécessairement exactement le nombre de classes souhaité, car un algorithme est utilisé pour déterminer ce nombre.

```
par(las = 1, cex.axis = 0.8, cex.lab = 0.8)
hist(diamonds$price, freq = F, col = "lightblue", nclass = 30,
     xlab = "Prix du diamant", main = "Distribution de la variable prix")
lines(density(diamonds$price), col = "red")
x_seq <- seq(-1000, 20000, by = 100)
lines(x_seq, dnorm(x_seq, mean(diamonds$price), sd(diamonds$price)))
abline(h = seq(0, 0.0005, by = 0.00005),
       v = seq(0, 20000, by = 2500),
       col = "lightgray", lty = "dotted")
```

Distribution de la variable prix



5.1.1.4 Nuage de points

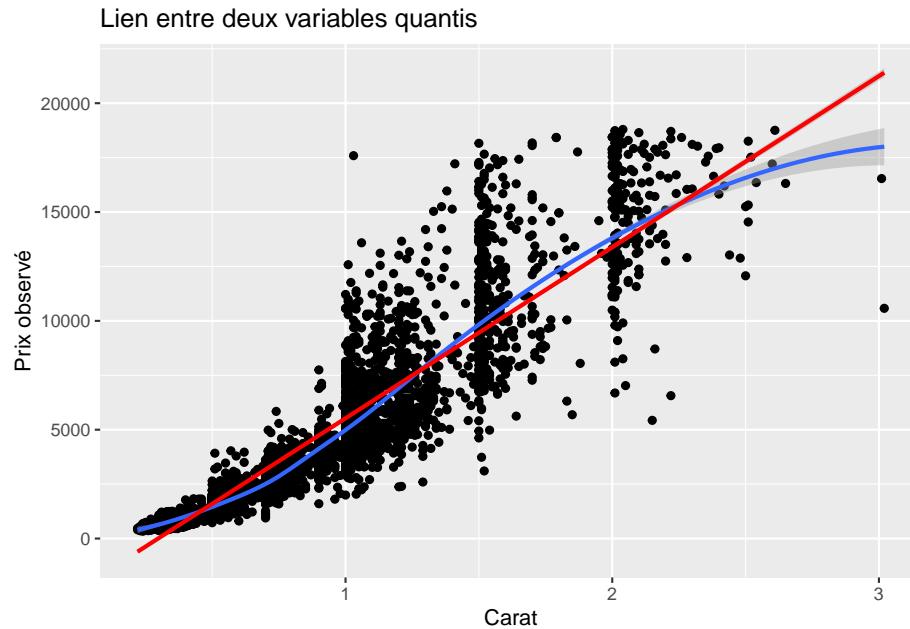
Afin d'avoir des représentations graphiques moins lourdes, on va se restreindre à un sous-échantillon de taille 5000. En effet, pour certains graphiques, lorsque le nombre d'observations est important, sa lecture devient difficile.

```
set.seed(123) # on fixe une graine aléatoire
diam_ech <- diamonds[sample(nrow(diamonds), 5000, replace = F), ]
```

Pour représenter un nuage de points et une droite de régression non paramétrique, cela se fait dans **ggplot2** avec les fonctions *geom_point()* et *geom_smooth()*. Cette dernière permet de tracer une droite de régression linéaire (option **method="lm"**) ou alors une droite de régression non paramétrique (option **method="loess"**), basée sur les modèles GAM (Hastie et Tibshirani, 1986). Un intervalle de confiance est également représenté par défaut pour cette dernière méthode.

```
ggplot(diam_ech) +
  aes(x = carat, y = price) + # on va chercher des variables dans diam_sample
  geom_point() + # on s'intéresse aux 2 variables carat et price
  geom_smooth(method = "loess") + # on ajoute une droite de rég. non paramétrique
  geom_smooth(method = "lm", # on ajoute une droite de régression linéaire
              col = "red") +
  xlab("Carat") +
  ylab("Prix observé") +
  ggtitle("Lien entre deux variables quantis")
```

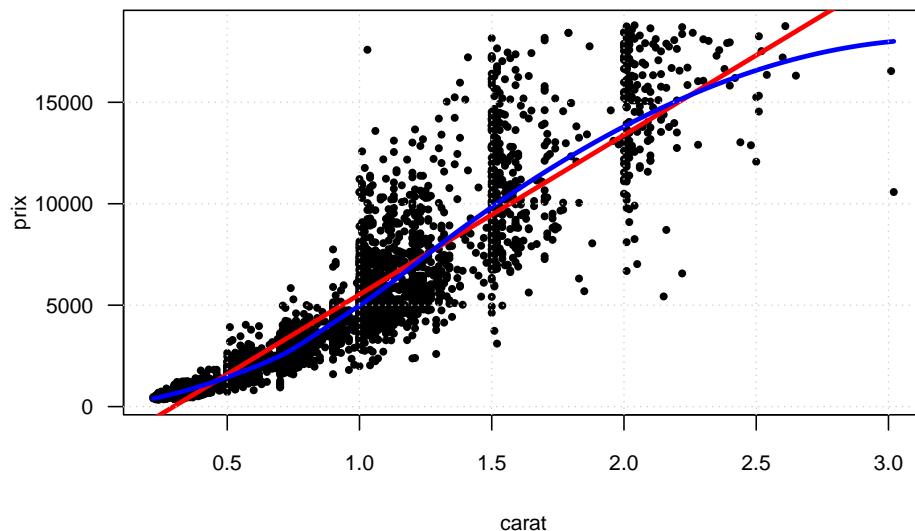
```
## `geom_smooth()` using formula 'y ~ x'
## `geom_smooth()` using formula 'y ~ x'
```



En utilisant les fonctions de base, il faut passer par les fonctions *lm()* pour calculer les coefficients de la droite de régression linéaire et la fonction *loess()* permet d'obtenir une droite de régression non paramétrique basée quant à elle sur la méthode des polynômes locaux. Pour représenter une droite de régression non paramétrique basée sur la méthode GAM, le lecteur pourra se référer au package **gam**.

```
par(las = 1, cex.axis = 0.8, cex.lab = 0.8)
plot(price ~ carat, data = diam_ech, pch = 16, cex = 0.7,
      xlab = "carat", ylab= "prix", main = "Lien entre deux variables quantis")
abline(lm(price ~ carat, data = diam_ech), col = "red", lwd = 3)
# values to predict
x_carat <- seq(0, 4.5, 0.01)
lines(x_carat,
      predict(loess(price ~ carat, data = diam_ech),
              data.frame(carat = x_carat)),
      col = "blue", lwd = 3)
abline(h = seq(0, 20000, by = 5000),
      v = seq(0, 4, by = 0.5),
      col = "lightgray", lty = "dotted")
```

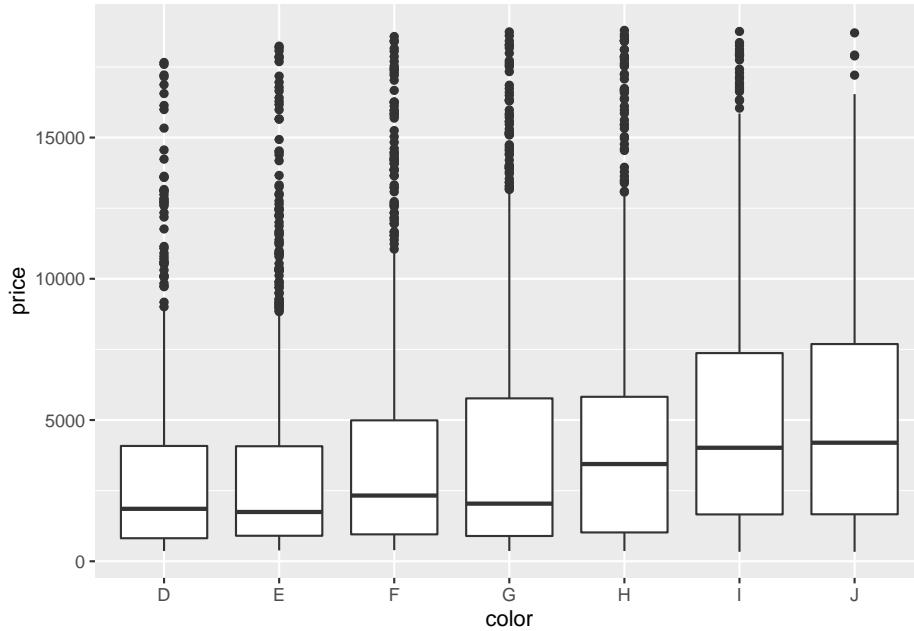
Lien entre deux variables quantis



5.1.1.5 Boîtes à moustaches parallèles

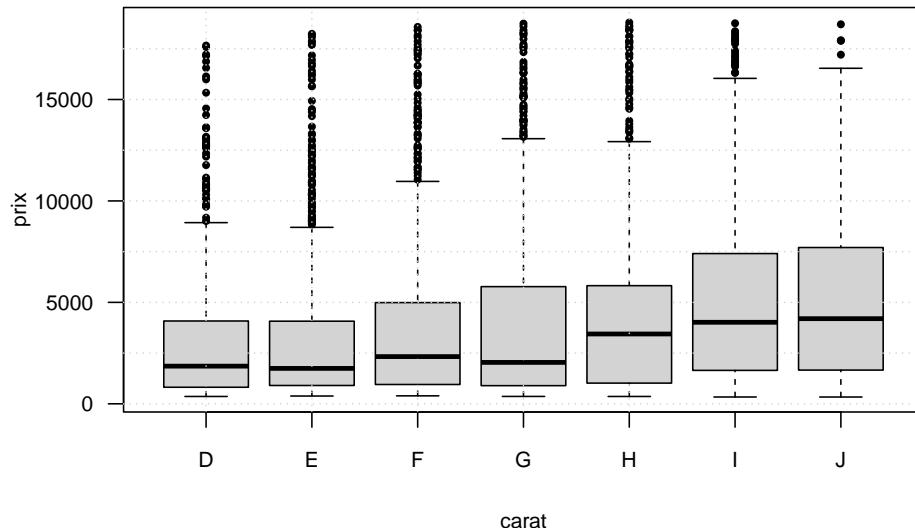
Avec **ggplot2**, les boîtes à moustaches parallèles se font à partir de la fonction `geom_boxplot()` :

```
ggplot(diam_ech) +  
  aes(x = color, y = price) +  
  geom_boxplot()
```



Pour obtenir un graphique similaire avec les fonctions de base, on peut utiliser la fonction *boxplot()* :

```
par(las = 1, cex.axis = 0.8, cex.lab = 0.8)
boxplot(price ~ color, data = diam_ech, pch = 16, cex = 0.7,
        xlab = "carat", ylab= "prix")
abline(h = seq(0, 20000, by = 2500),
       v = seq(0, 5, by = 1),
       col = "lightgray", lty = "dotted")
```



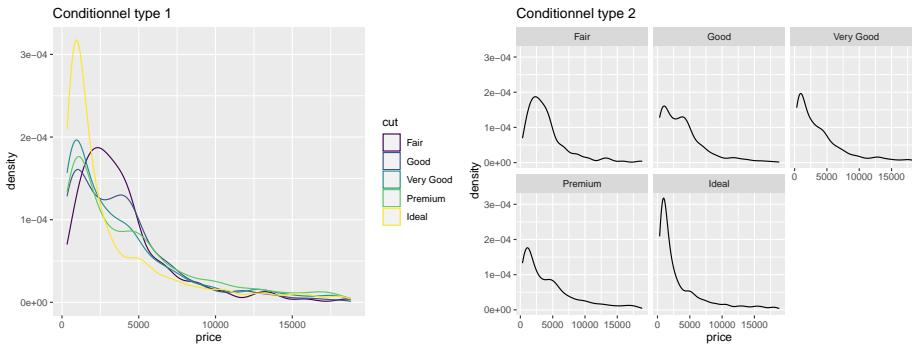
Conclusion : pour des graphiques “standards”, on constate donc que le match “fonctions de base” VS “fonctions **ggplot2**” n’a pas vraiment de gagnants. Par ailleurs, ici nous sommes restés sur des graphiques simples avec un minimum d’ornementations. Le package **ggplot2** repose sur une représentation de données de type **data.frame**; du coup, pour faire certains ajouts tels que des points particuliers, des lignes brisées ou des étiquettes, cela peut s’avérer plus compliqué que d’utiliser simplement les fonctions graphiques de base. Malheureusement, il n’est pas possible de combiner les fonctions graphiques de base avec les fonctions du package **ggplot2**.

5.1.2 Graphiques conditionnels

Un avantage du package **ggplot2** sur les fonctions graphiques de base repose sur la très bonne gestion des graphiques conditionnels. Les graphiques conditionnels correspondent à des graphiques standards qu’on va réaliser conditionnellement à une variable souvent qualitative.

Il y a deux façons de faire un graphique conditionnel :

- on va afficher dans un même graphique une information relative aux modalités d’une variable qualitative (ex : dans un nuage de points, on va représenter une droite de régression associée à chacune des modalités d’une variable qualitative).
- on va faire autant de graphiques qu’il existe de modalités d’une variable qualitative (ex : on va faire un histogramme d’une variable quantitative pour chaque modalité d’une variable qualitative).

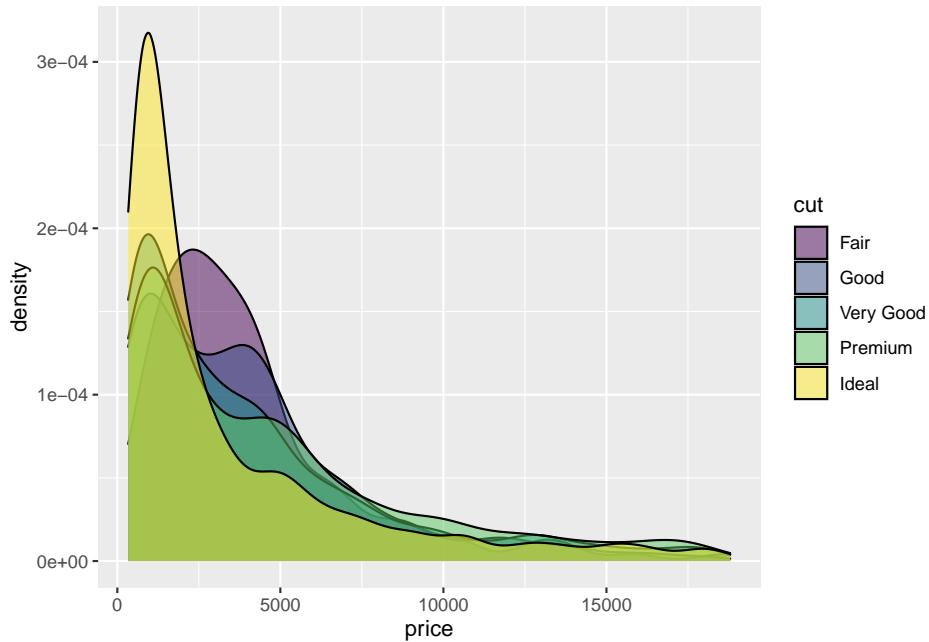


5.1.2.1 Utilisation des options `fill=` et `colour=`

L'utilisation des options `fill=` et `colour=` à l'intérieur de la fonction `aes()` ou des fonctions de type `geom_XXX()`, vont permettre de réaliser le premier type de graphique.

5.1.2.1.1 Fonctions de densités On souhaite savoir si la distribution de la `price` est la même selon les modalités prises par la variable qualitative `cut`. Pour ce faire, on va ajouter l'option `fill=` dans la fonction `aes()` suivie du nom de la variable qualitative. L'option `fill=` permet de remplir chaque courbe de densité par une couleur prédéfinie dans **ggplot2**. Si on avait utilisé l'option `colour=`, on aurait obtenu uniquement des traits de couleurs différentes. L'option `alpha=` permet quant à elle de définir le degré de transparence des couleurs.

```
ggplot(diam_ech) +
  aes(x = price, fill = cut) +
  geom_density(alpha = 0.5)
```



Pour réaliser un tel graphique en utilisant les fonctions de base, cela nécessite de réaliser les étapes suivantes :

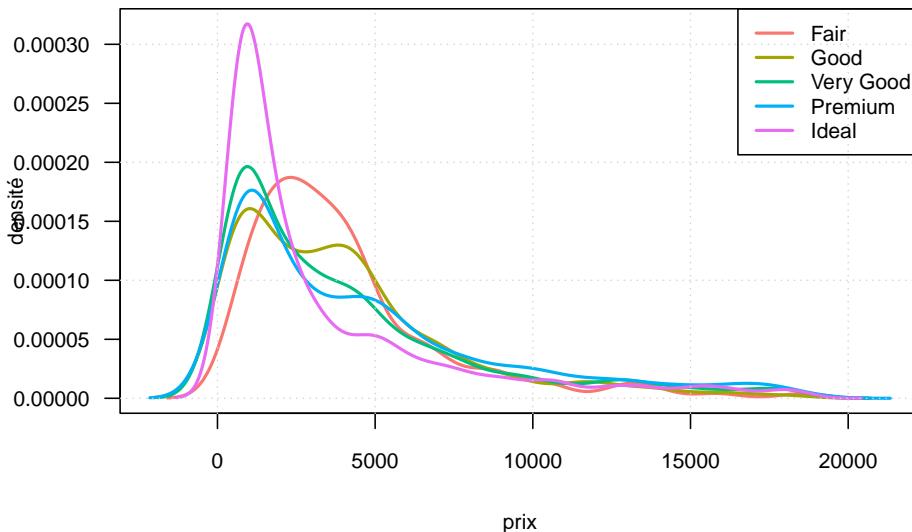
- découper la variable quantitative en fonction des modalités, ceci pouvant se faire avec la fonction *split()*,
- appliquer la fonction *density()* sur chaque sous-échantillon,
- ouvrir une fenêtre graphique en définissant correctement les marges,
- représenter une courbe de densité par modalité en utilisant une couleur pour chaque courbe à définir (pour obtenir les mêmes couleurs que celles obtenues proposées dans **ggplot2**, on pourra voir ce lien : <https://stackoverflow.com/questions/8197559/emulate-ggplot2-default-color-palette>).

```
# on split la variable price en fonction de carat
list_price <- split(diam_ech$price, diam_ech$cut)
# on applique la fonction density à chaque sous-groupe
list_density <- lapply(list_price, density)
# on ouvre la fenêtre graphique avec les paramètres optimaux
par(las = 1, cex.axis = 0.8, cex.lab = 0.8)
plot(range(unlist(lapply(list_density, function(l) range(l$x)))),
     range(unlist(lapply(list_density, function(l) range(l$y)))),
     type = "n",
     xlab = "prix",
     ylab = "densité")
# choix d'une palette de couleurs
```

```

col_pal <- c("#F8766D", "#A3A500", "#00BF7D", "#00B0F6", "#E76BF3")
# application de la fonction lines() sur chaque élément de la liste
dont_print <- mapply(lines, list_density,
  col = col_pal, lwd = 2)
abline(h = seq(0, 4*10^(-4), by = 10^(-4)),
  v = seq(0, 25000, by = 5000),
  col = "lightgray", lty = "dotted")
legend("topright", legend = names(list_density),
  col = col_pal, lwd = 2, cex = 0.8)

```



Conclusion : on voit donc que **ggplot2** a très bien pris en compte la représentation de graphiques conditionnellement à une variable qualitative. Un autre atout est la gestion des paramètres graphiques par défaut (de type couleur, légende, etc). C'est pour ces deux raisons que **ggplot2** rencontre un tel succès. Il est important de souligner que derrière l'utilisation de toutes ces fonctions, se cachent des tas de lignes de codes qui rendent possibles cette utilisation simplifiée.

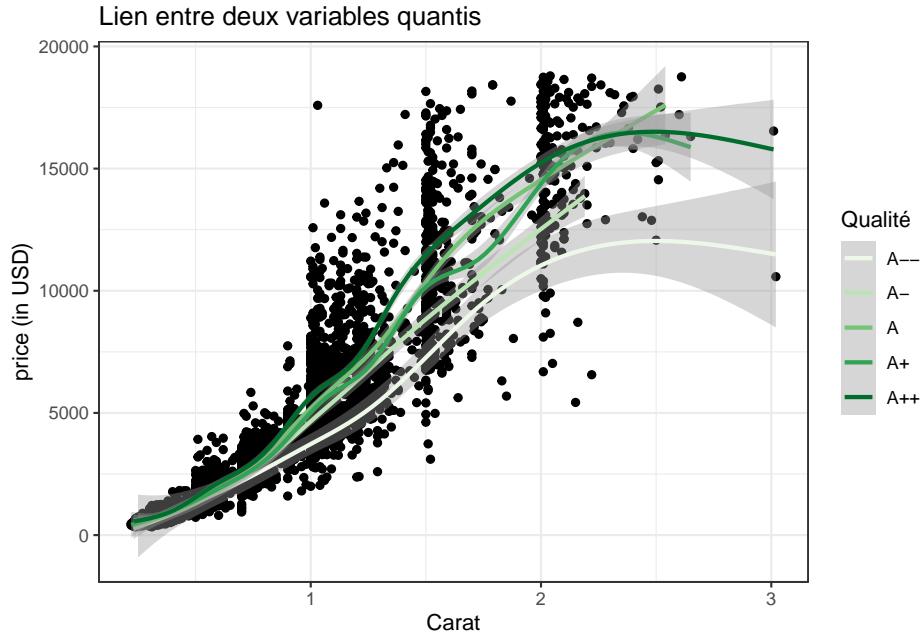
Remarque : pour remplir les aires sous les courbes de densités avec les graphiques de base, on aurait pu utiliser la fonction *polygon()*

5.1.2.1.2 Nuages de points L'utilisation d'une variable qualitative conditionnelle peut s'avérer très intéressante à utiliser sur un nuage de points, pour savoir notamment si le lien entre les variables quantitatives Y et X sont les mêmes en fonction d'une variable qualitative. Ici on a changé la palette de couleur en utilisant les palettes de couleurs inspirés du package **RColorBrewer** (voir la note suivante concernant les palettes de couleurs style **RColorBrewer** : <https://www.nceas.ucsb.edu/~frazier/RSpatialGuides/colorPaletteCheatshee>

t.pdf).

```
ggplot(diam_ech) +
  aes(x = carat, y = price) +
  geom_point() +
  geom_smooth(aes(colour = cut)) + # ajoute 5 courbes selon la variable cut
  theme_bw() + # modifie la couleur de fond
  xlab("Carat") + # modifie la légende de l'axe des x
  ylab("price (in USD)") + # modifie la légende de l'axe des y
  ggtitle("Lien entre deux variables quantis") + # ajoute un titre
  scale_colour_brewer(name = "Qualité", # modifie la légende de cut
                       labels = c("A--", "A-", "A", "A+", "A++"), # Etiquette
                       palette = "Greens") # modifie la palette de couleurs

## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



Pour réaliser un graphique équivalent avec les fonctions graphiques de base, il faut procéder comme on l'a fait dans l'exemple précédent. On splité dans un premier temps le **data.frame** en fonction de la variable qualitative, on applique ensuite les fonctions *predict()* et *loess()* aux sous-échantillons avant de représenter les courbes les unes après les autres en utilisant les couleurs choisies.

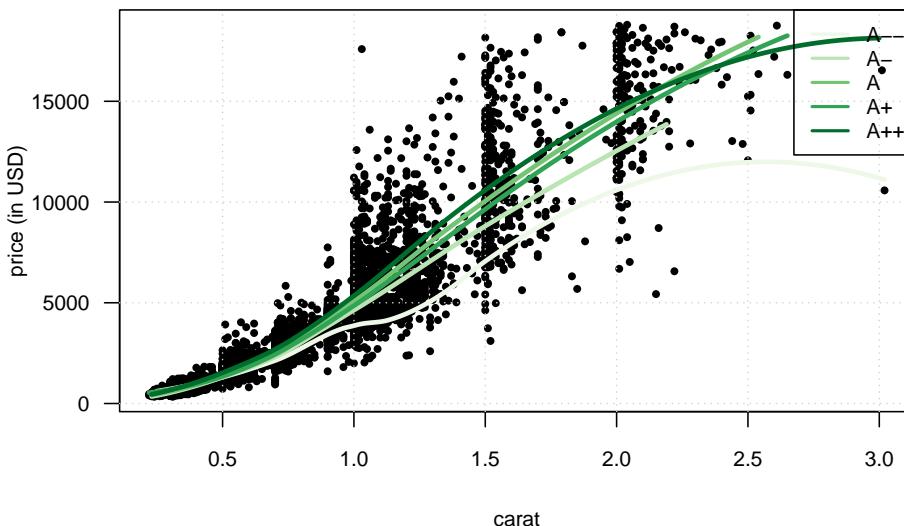
```
par(las = 1, cex.axis = 0.8, cex.lab = 0.8)
plot(price ~ carat, data = diam_ech, pch = 16, cex = 0.7,
      xlab = "carat", ylab = "price (in USD)",
      main = "Lien entre deux variables quantis")
abline(h = seq(0, 20000, by = 5000),
```

```

v = seq(0, 4, by = 0.5),
      col = "lightgray", lty = "dotted")
# on split le data.frame en fonction de cut
list_df <- split(diam_ech, diam_ech$cut)
# values to predict
x_carat <- seq(0, 4.5, 0.01)
# on applique la fonction lowess à chaque sous-groupe
list_loess <- lapply(list_df, function(obj)
  predict(loess(price ~ carat, data = obj),
         data.frame(carat = x_carat)))
# choix d'une palette de couleurs
require("RColorBrewer")
col_pal <- brewer.pal(length(list_price), "Greens")
# application de la fonction lines() sur chaque élément de la liste
dont_print <- mapply(lines, rep(list(x_carat), 5), list_loess,
  col = col_pal, lwd = 3)
legend("topright", legend = c("A--", "A-", "A", "A+", "A++"),
  col = col_pal, lwd = 2, cex = 0.8)

```

Lien entre deux variables quantis

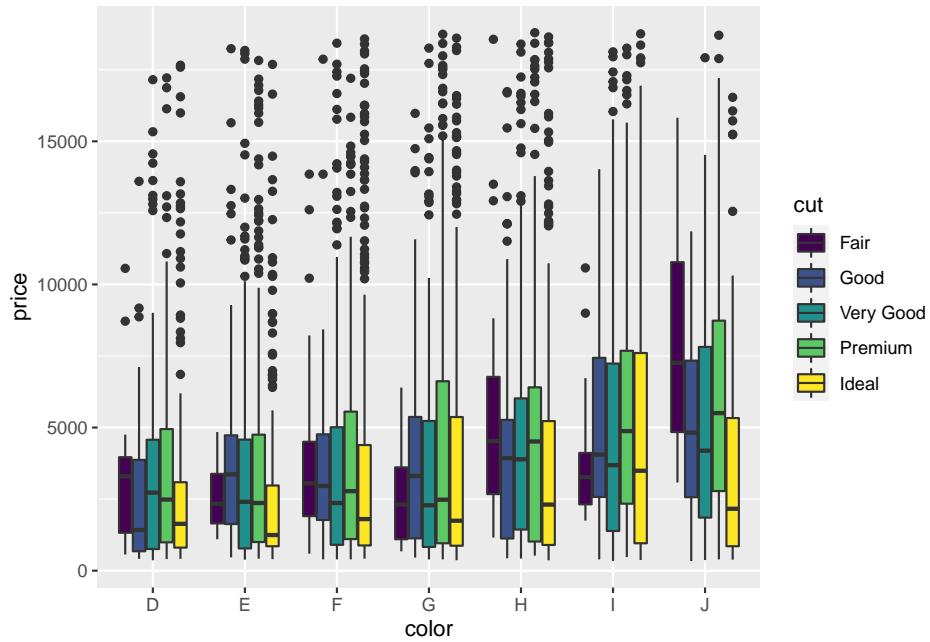


5.1.2.1.3 Boîtes à moustaches Pour réaliser des boîtes à moustache parallèles conditionnellement à une variable qualitative, cela se fait en seulement 3 lignes de codes avec **ggplot2**.

```

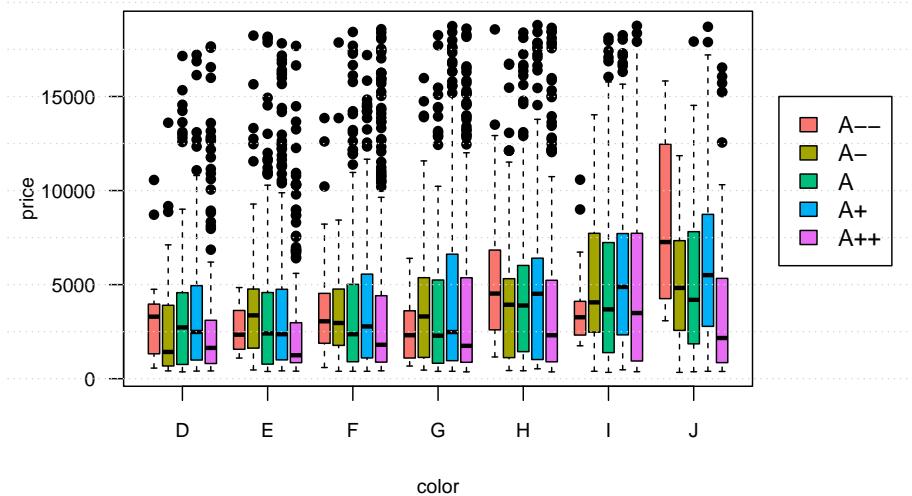
ggplot(diam_ech) +
  aes(x = color, y = price, fill = cut) +
  geom_boxplot()

```



La fonction par défaut `boxplot()` permet de faire quelque chose de similaire en paramétrant suffisamment bien les options. Pour afficher la boîte de légendes en-dehors du cadre, on a utilisé l'option `xpd=T` et modifié le paramètre de la marge à droite.

```
# choix des couleurs :
col_pal <- c("#F8766D", "#A3A500", "#00BF7D", "#00B0F6", "#E76BF3")
par(las = 1, cex.axis = 0.8, cex.lab = 0.8,
    xpd = T, mar = par()$mar + c(0, 0, 0, 4))
boxplot(price ~ cut + color, data = diam_ech,
        xlab = "color", ylab = "price",
        at = c(1:5, 7:11, 13:17, 19:23, 25:29, 31:35, 37:41),
        col = rep(col_pal, 7),
        pch = 16,
        xaxt = "n")
axis(1, at = c(3, 9, 15, 21, 27, 33, 39),
      labels = c("D", "E", "F", "G", "H", "I", "J"))
abline(h = seq(0, 20000, by = 2500), col = "lightgray", lty = "dotted")
legend(45, 15000, legend = c("A--", "A-", "A", "A+", "A++"),
       fill = col_pal)
```

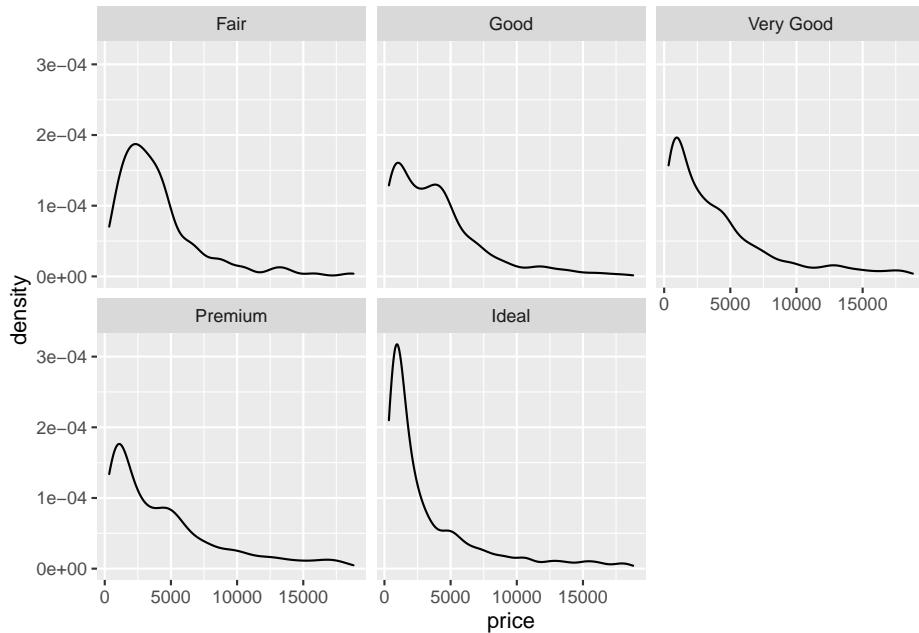


5.1.2.2 Utilisation des facet

L'utilisation des fonctions de type `facet_XXX()` dans **ggplot2** permet de créer autant de fenêtre graphiques qu'il y a de modalités dans la variable conditionnelle.

Si on reprend l'exemple précédent de la distribution conditionnelle de la variable **price** en fonction de la variable **cut**, plutôt que de représenter toutes les distributions dans la même fenêtre graphique, on peut vouloir représenter chaque distribution dans une fenêtre graphique différente. Cela se fait avec les fonctions de type `facet_XXX()`.

```
ggplot(diam_ech) +
  aes(x = price) +
  geom_density() +
  facet_wrap(~ cut)
```



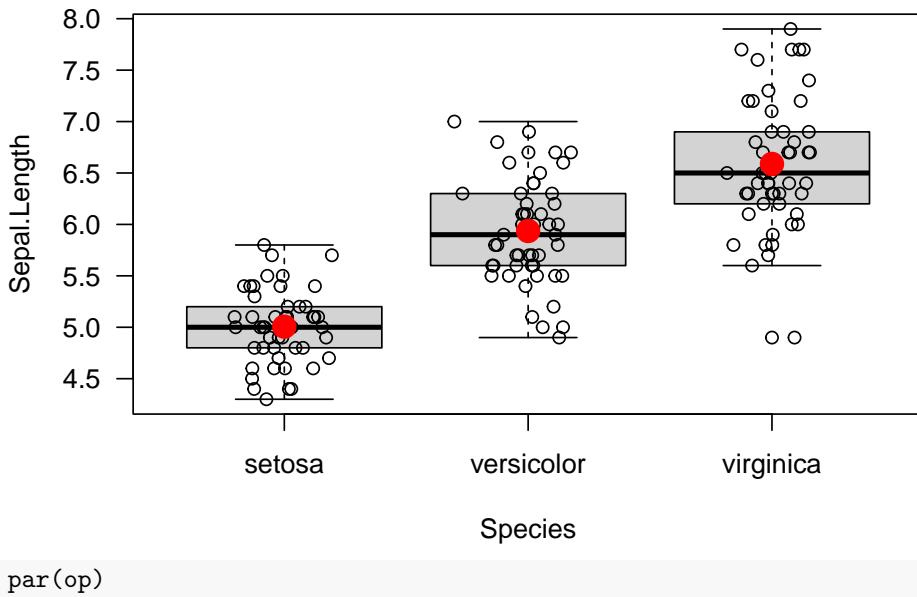
Réaliser le type de graphique ci-dessus avec les fonctions de base de **R** devient extrêmement compliqué : il faut d'abord découper la fenêtre en 6, puis utiliser une boucle **for** pour appeler la fonction *plot()* dans chaque sous-compartiment, etc.. Avant l'arrivée du package **ggplot2**, le package **lattice** qui fait partie des packages de base, permet également de faire des graphiques conditionnels en quelques lignes de codes. Ici, nous ne présentons pas d'exemples d'utilisation de ce package, mais le lecteur pourra trouver des exemples de comparaison entre **ggplot2** et **lattice** dans ce document :

- https://www.londonr.org/wp-content/uploads/sites/2/presentations/LondonR_-_lattice_vs_ggplot2_-_Richard_Pugh_and_Any_Nicholls_-_20130910.pdf

Exercice 4.1

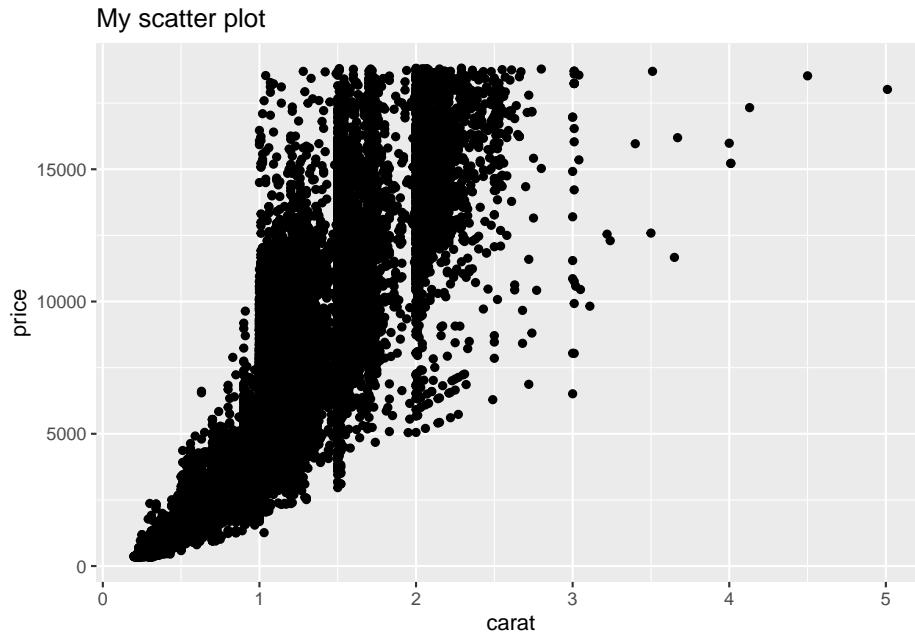
- Trouver le code en syntaxe **ggplot** qui permette d'obtenir le graphique suivant obtenu en syntaxe de base:

```
op <- par(oma = c(1, 1, 0, 1), las = 1)
boxplot(Sepal.Length ~ Species, data = iris)
points(as.numeric(iris$Species) + rnorm(150, 0, 0.1), iris$Sepal.Length)
points(c(1, 2, 3), tapply(iris$Sepal.Length, iris$Species, mean),
      col = "red", pch = 16, cex = 2)
```



- Trouver le code en syntaxe de base qui permette d'obtenir le graphique suivant obtenu avec la syntaxe **ggplot**:

```
data("diamonds")
ggplot(diamonds,
       aes(x = carat,
            y = price)) +
  geom_point() +
  ggtitle("My scatter plot")
```



5.2 Présentation de graphiques originaux

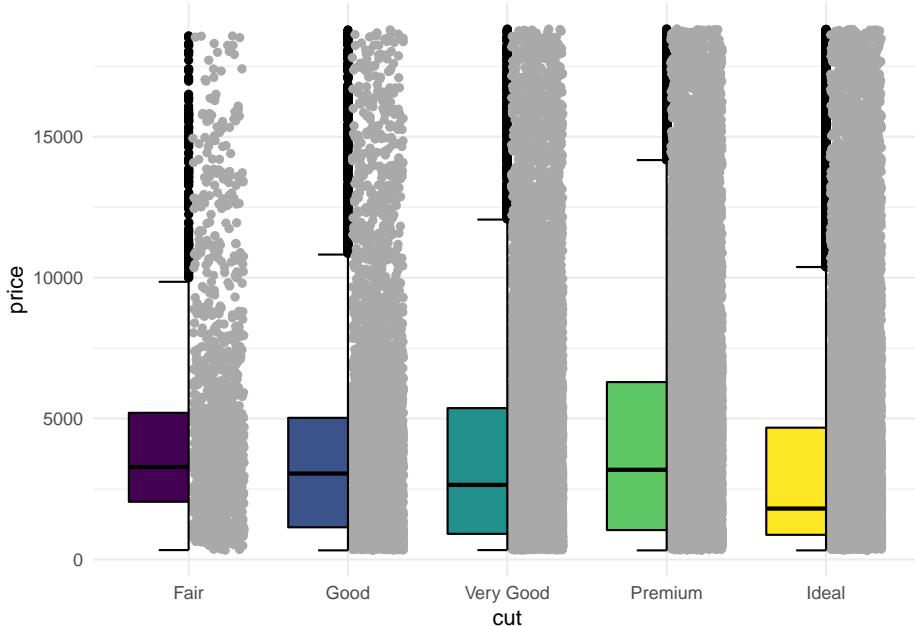
On présente ici une liste de graphiques originaux, la plupart inspirée de la syntaxe **ggplot2**, mais pas uniquement.

5.2.1 Mélange boîte à moustache/diagramme de dispersion

Le graphique suivant (issu du package **ggbpol**) permet de représenter une “demi” boîte à moustache et un diagramme de dispersion à la place de la demi boîte. Le diagramme de dispersion n'est pas représenté sur une unique droite. En effet, si plusieurs valeurs sont identiques, il n'est pas possible de les distinguer sur une droite. C'est pourquoi les données ne sont pas toutes représentées sur la même droite (elles sont **jitter** en anglais). L'intérêt de ce graphique est limité, mais il permet toutefois d'apprécier le volume de données correspondant à chaque partie de la boîte à moustache.

```
library("ggbpol")
ggplot(diamonds,
       aes(x = cut,
           y = price,
           fill = cut)) +
  geom_boxjitter(color = "black",
                 jitter.color = "darkgrey",
                 errorbar.draw = TRUE) +
```

```
theme_minimal() +
  theme(legend.position = "none")
```



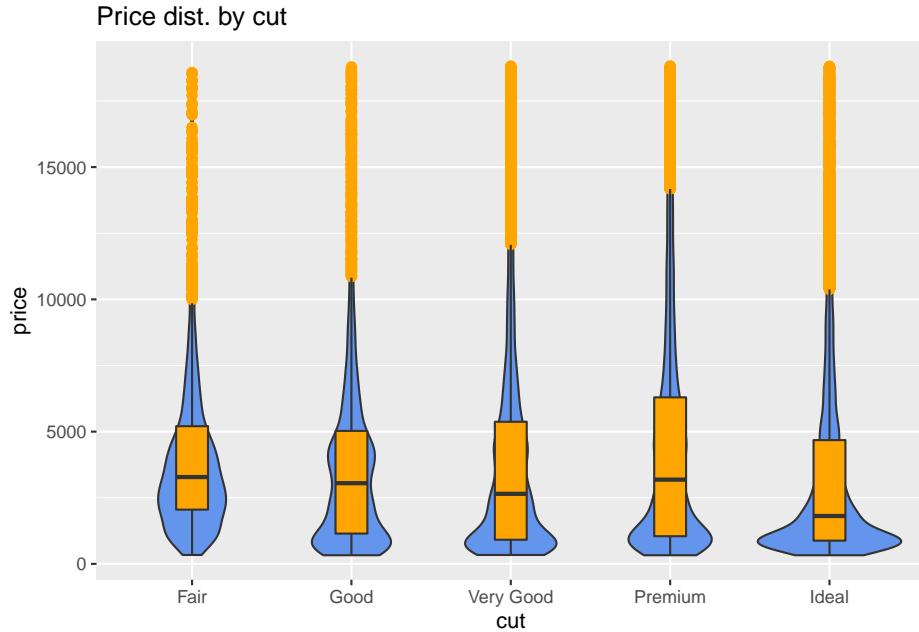
5.2.2 Violin plot

Le but de ce graphique est de représenter sur un même graphique une boîte à moustache et un estimateur non paramétrique de la densité. L'objectif est de visualiser sur un même graphique les informations spécifiques apportées par l'un et par l'autre. Plus précisément la boîte à moustache est intéressante pour visualiser les valeurs extrêmes alors que l'estimateur non paramétrique de la densité permet d'avoir une idée de l'allure de la distribution.

L'estimation non paramétrique est représentée verticalement (alors que celle-ci est généralement représentée horizontalement); de plus, il y a un effet miroir dans le but d'harmoniser le dessin avec la boîte à moustache. Dans l'exemple ci-dessous, on a représenté le “violin plot” de la variable **price** en fonction des modalités de la variable **cut**.

```
ggplot(diamonds,
       aes(x = cut,
           y = price)) +
  geom_violin(fill =
               "cornflowerblue") +
  geom_boxplot(width = .2,
               fill = "orange",
               outlier.color = "orange",
```

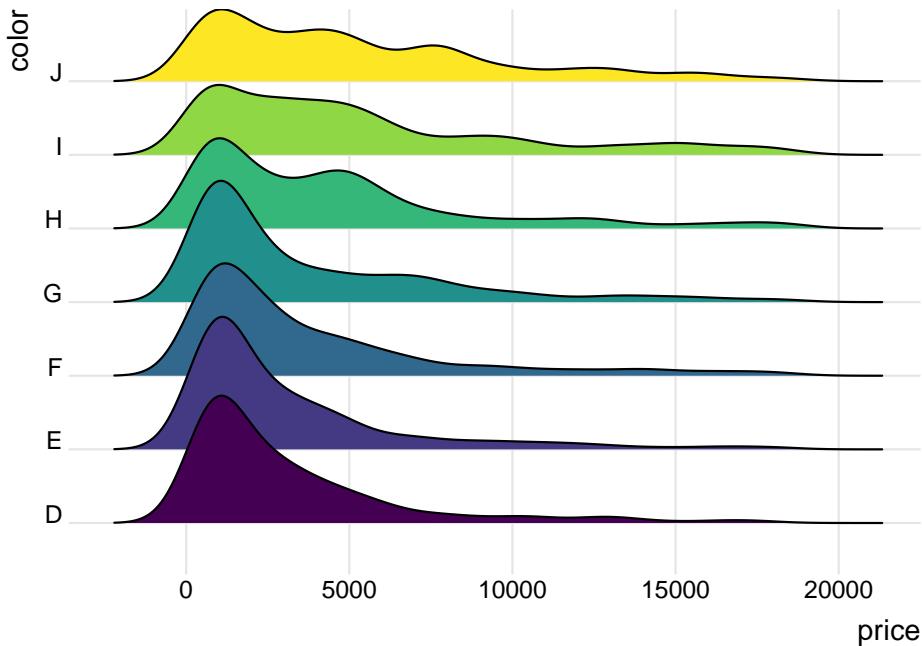
```
outlier.size = 2) +
  labs(title = "Price dist. by cut")
```



5.2.3 Ridgeline plots

Ce graphique (issu du package `ggridges`) est particulièrement intéressant car il permet de visualiser des estimateurs non paramétriques de densité conditionnement à une variable qualitative. Contrairement à un des graphiques vue dans la section précédente, il est possible d'apprécier correctement la forme de chaque estimateur car ceux-ci sont représentés sur une ligne différente, tout en conservant la même échelle.

```
library(ggridges)
ggplot(diam_ech,
       aes(x = price,
           y = color,
           fill = color)) +
  geom_density_ridges() +
  theme_ridges() +
  labs("Price by levels color") +
  theme(legend.position = "none")
```



5.2.4 Barres d'erreurs

Le graphique suivant permet de représenter sur un graphique les barres d'erreurs correspondant aux intervalles de confiance à 95% autour des moyennes observées, en utilisant l'hypothèse de normalité. Attention, cela ne remplace pas un test statistique, mais cela permet d'apprecier visuellement si deux moyennes observées semblent significativement différentes l'une de l'autre.

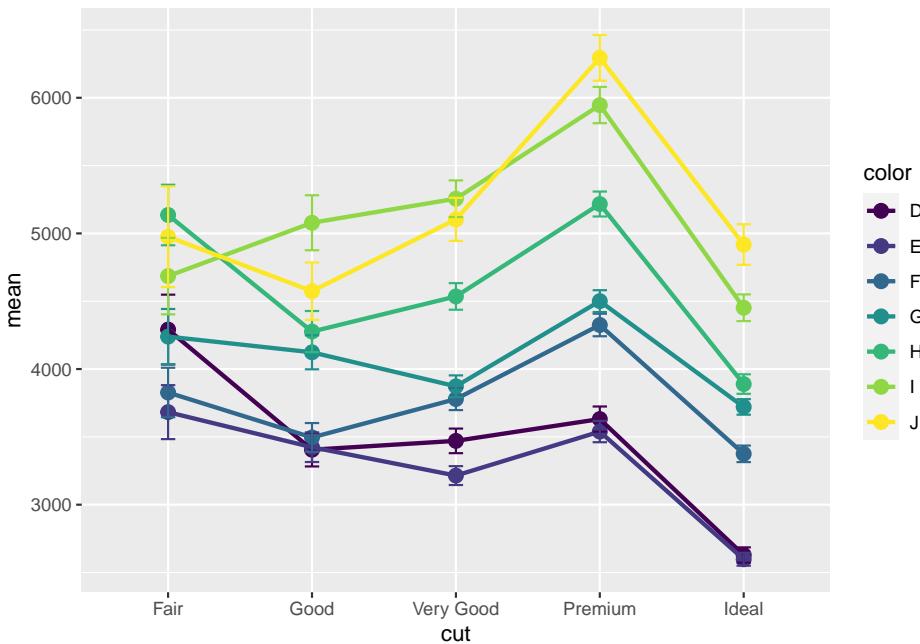
Pour réaliser ce graphique, il faut d'abord mettre en forme les données à la **ggplot2**, c'est-à-dire sous forme de **data.frame**, correctement disposé (**tidy**).

```
library(dplyr)
plotdata <- diamonds %>%
  group_by(color, cut) %>%
  summarize(n = n(),
            mean = mean(price),
            sd = sd(price),
            se = sd / sqrt(n),
            ci = qt(0.975, df = n - 1) * sd / sqrt(n))
```

Ensuite, on peut représenter les différentes moyennes et les barres d'erreurs.

```
ggplot(plotdata, aes(x = cut,
                      y = mean,
                      group = color,
                      color = color)) +
```

```
geom_point(size = 3) +
  geom_line(size = 1) +
  geom_errorbar(aes(ymin = mean - se,
                     ymax = mean + se),
                width = .1)
```



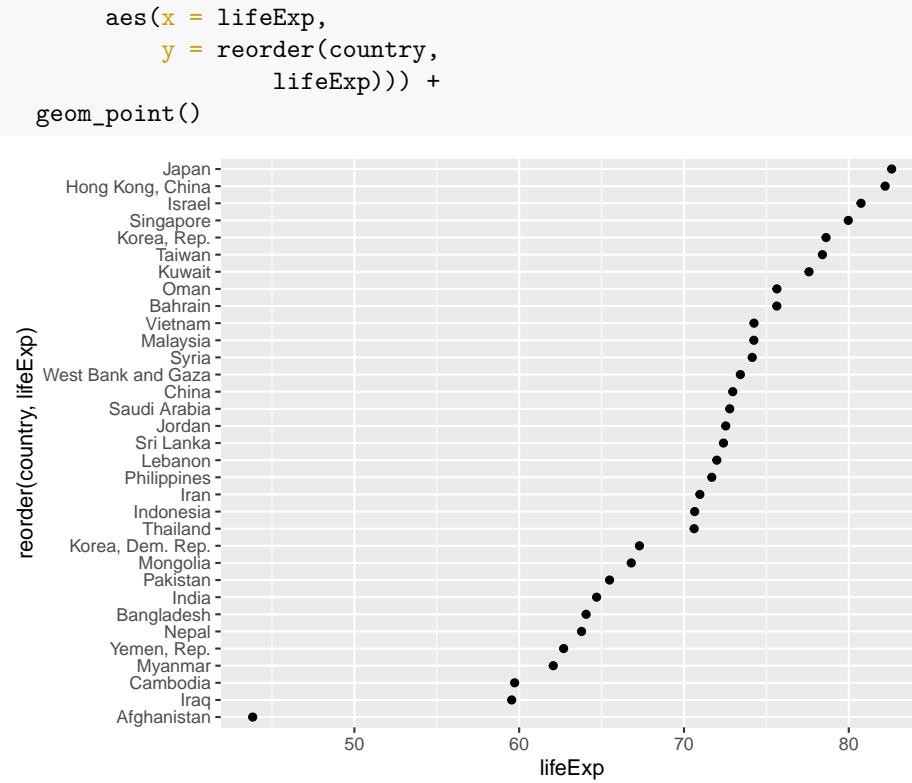
5.2.5 Cleveland dot chart

Le Cleveland dot chart est un graphique qui permet de visualiser une variable quantitative de façon brute. On représente en général en ordonnées l'identifiant d'une observation (comme une marque de voiture, un pays, etc.) et en abscisse la valeur observée. Cet outil est particulièrement intéressant lorsque les observations sont triées. Cela permet d'apprécier un ordre de grandeur des différences observées.

Dans l'exemple ci-dessous, les données proviennent du package **gapminder** sur les espérances de vie observées dans les pays d'Asie en 2007.

```
data(gapminder, package = "gapminder")

library(dplyr)
plotdata <- gapminder %>%
  filter(continent == "Asia" &
         year == 2007)
ggplot(plotdata,
```



5.2.6 Area chart

Ce graphique est intéressant pour représenter des données de composition (autrement dit des variables dont la somme est constante) qui évoluent au cours du temps. Dans l'exemple ci-dessous, on a représenté des parts de marché qui évoluent au cours du temps.

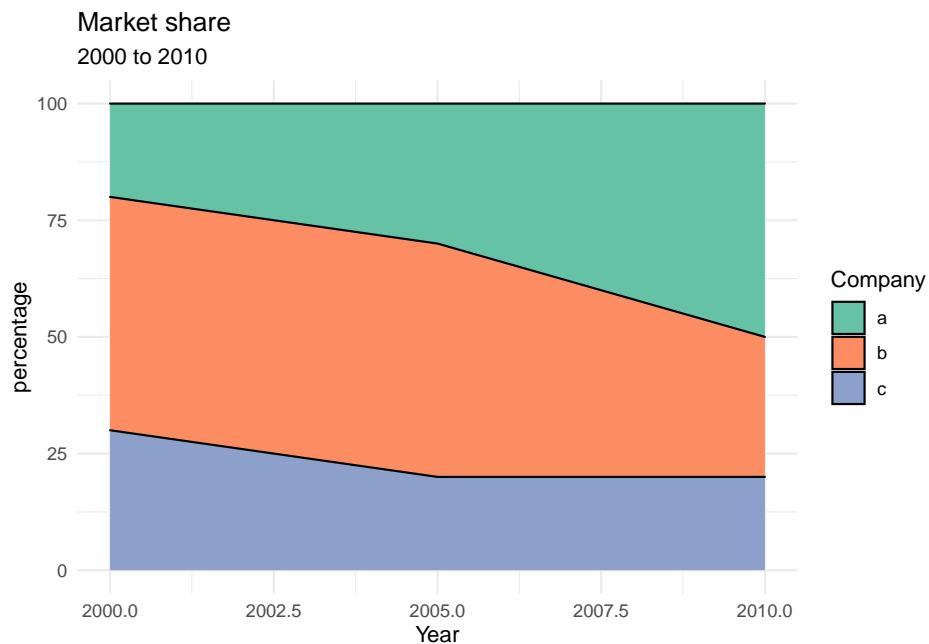
```

time_chart <- data.frame(
  year = rep(c(2000, 2005, 2010), each =3),
  market_share = c(20, 50, 30,
                  30, 50, 20,
                  50, 30, 20),
  comp = rep(c("a", "b", "c"), 3)
)

ggplot(time_chart,
aes(x = year,
y = market_share,
fill = comp)) +
geom_area(color = "black") +

```

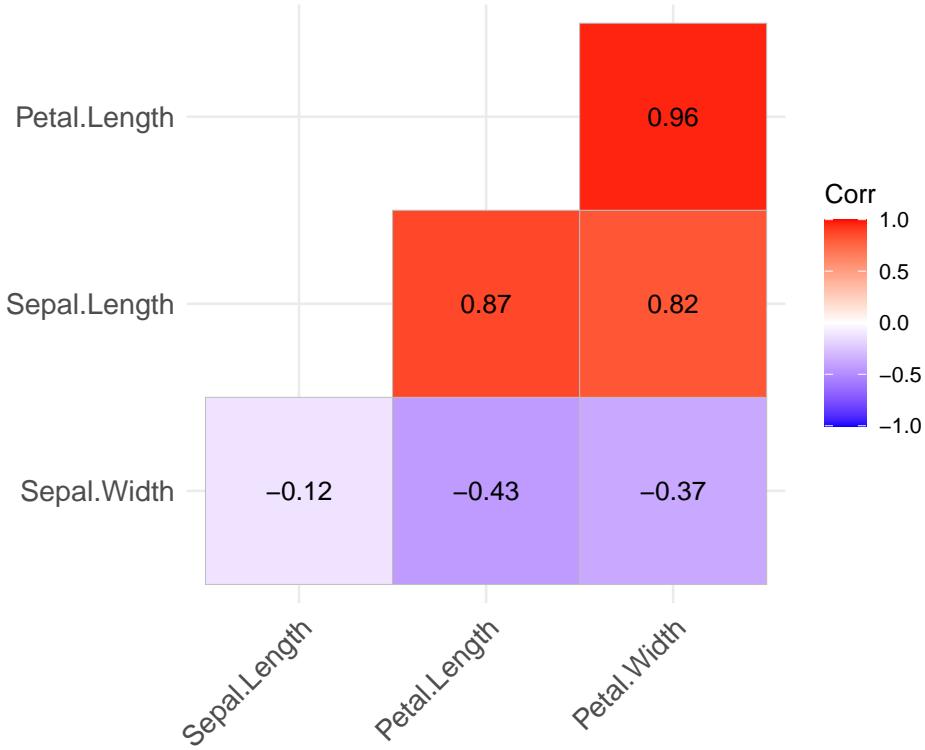
```
labs(title = "Market share",
     subtitle = "2000 to 2010",
     x = "Year",
     y = "percentage",
     fill = "Company") +
scale_fill_brewer(palette = "Set2") +
theme_minimal()
```



5.2.7 Correlation plot

Le graphique de corrélation permet d’apprécier le degré de corrélation entre plusieurs variables. Ce graphique provient du package **ggcorrplot** et a été profondément épuré. On en effet, une matrice de corrélation est symétrique et constante sur la diagonale; par conséquent, on ne peut garder que l’information provenant de la partie supérieure (ou inférieure). C’est ce que fait le graphique suivant.

```
library(ggcorrplot)
r <- cor(iris[, 1:4], use = "complete.obs")
ggcorrplot(r,
           hc.order = TRUE,
           type = "lower",
           lab = TRUE)
```



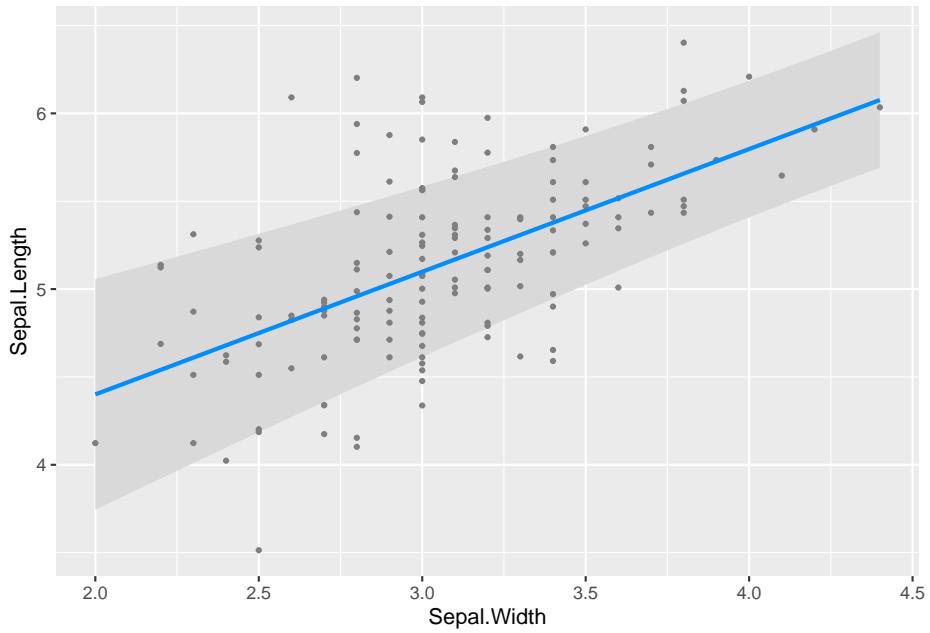
5.2.8 Représenter les effets des variables explicatives

Dans ce paragraphe, nous allons voir des graphiques qui permettent de représenter certaines informations provenant de modèles statistiques.

5.2.8.1 Régression linéaire

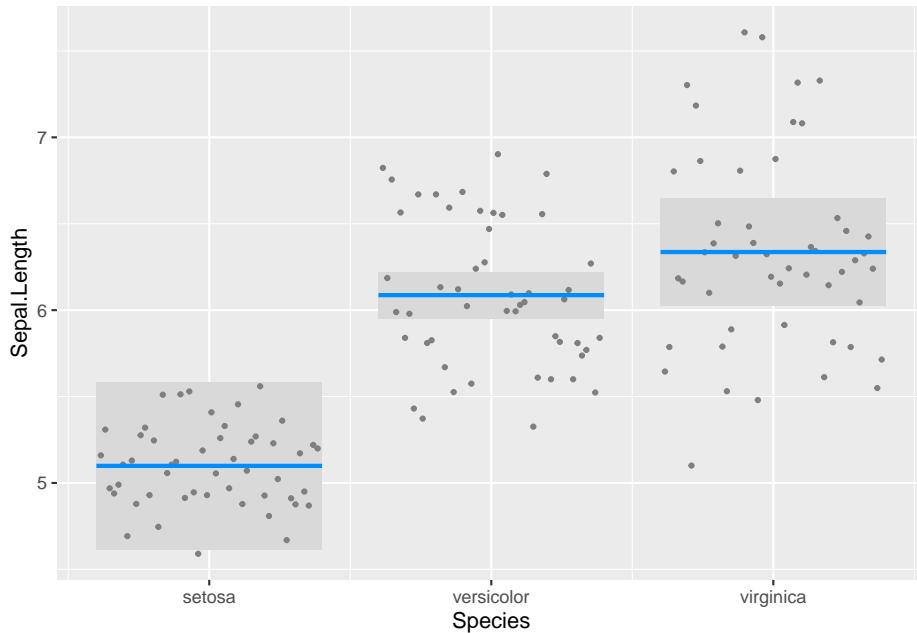
La fonction `visreg()` du package `visreg` permet de représenter les effets d'une variable explicative sur la variable à expliquer. Pour cela, on représente sur le nuage de points les valeurs prédictes avec une doite bleue. On notera que pour calculer les valeurs prédictes à partir du modèle estimé, les autres variables du modèle sont prises constantes et égales à la médiane pour les variables quantitatives et au mode pour les variables qualitatives (plus de détails sur : <https://journal.r-project.org/archive/2017/RJ-2017-046/RJ-2017-046.pdf>).

```
library(visreg)
res_lm <- lm(Sepal.Length ~
  Sepal.Width + Petal.Width + Species,
  data = iris)
visreg(res_lm, "Sepal.Width",
  gg = TRUE)
```



Pour les variables qualitatives, les valeurs prédites sont représentées par des traits bleus sur des diagrammes de dispersion conditionnels aux modalités de la variable explicative.

```
visreg(res_lm, "Species",
       gg = TRUE)
```

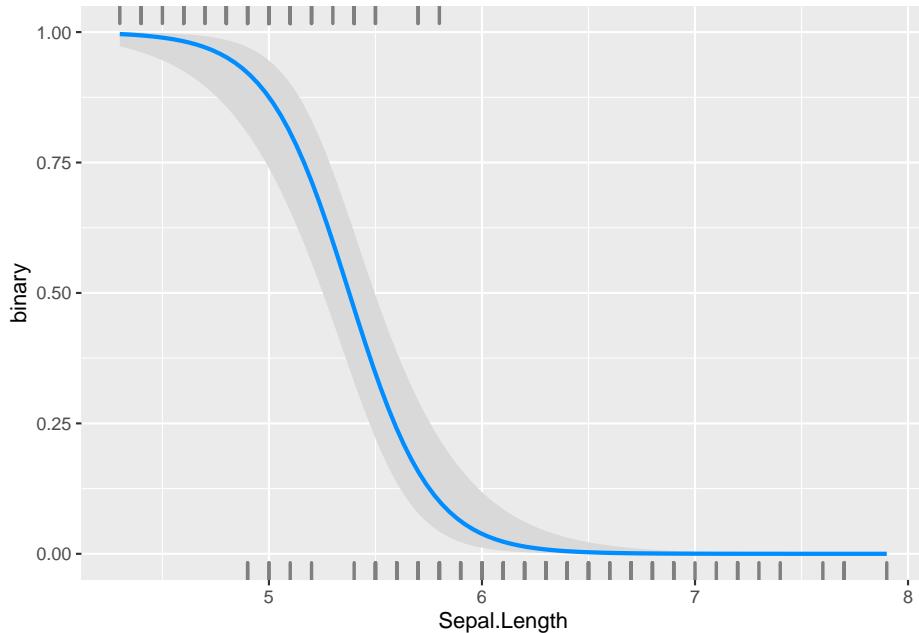


5.2.8.2 Régression logistique

Une façon de représenter les résultats d'une régression logistique est de représenter la probabilité que l'événement d'intérêt $Y = 1$ arrive, conditionnellement à une variable explicative. C'est ce que fait la fonction `visreg()` dans un modèle de régression logistique.

```
iris$binary <- factor(ifelse(iris$Species == "setosa", 1, 0))
res_glm <- glm(binary ~
  Sepal.Length,
  family = binomial(link = "logit"),
  data = iris)

visreg(res_glm, "Sepal.Length",
       gg = TRUE,
       scale="response")
```

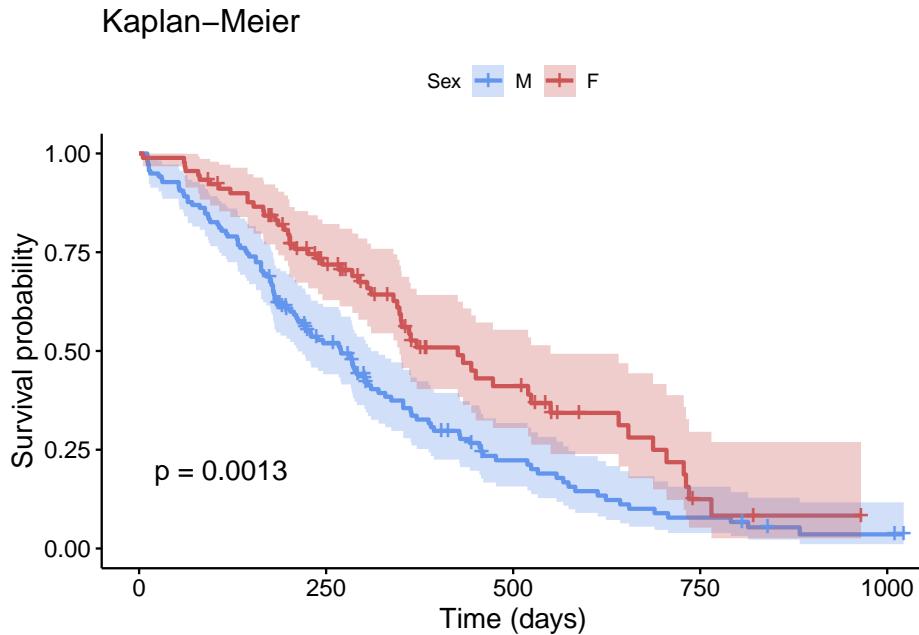


5.2.8.3 Modèle de survie

Dans un modèle de survie, on peut être intéresser de représenter la fonction estimée de survie en fonction du temps et conditionnellement à une variable qualitative (dans l'exemple ci-dessous le sexe). C'est ce que fait le package **survminer**. Le package **survival** permet de faire les estimations du modèle.

```
library(survival)
library(survminer)
data(lung)

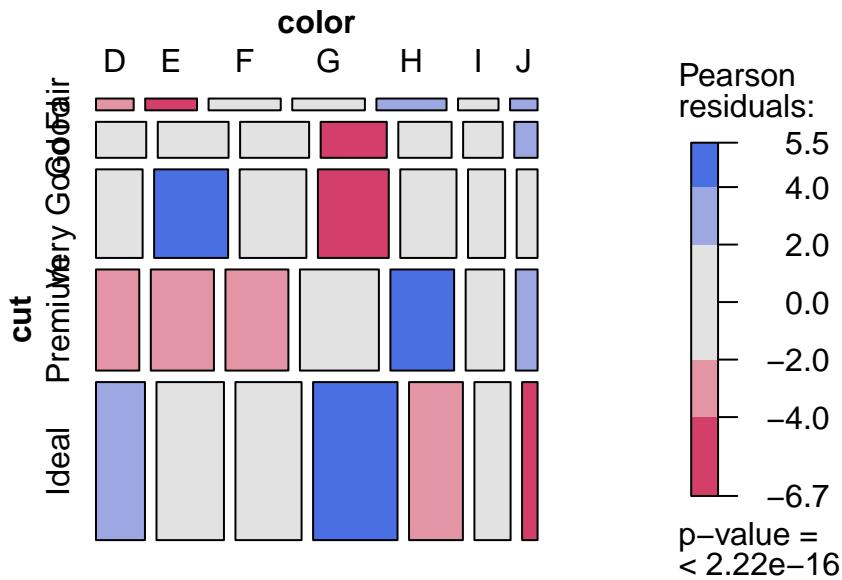
## Warning in data(lung): data set 'lung' not found
sfit <- survfit(Surv(time, status) ~ sex, data=lung)
ggsurvplot(sfit,
  conf.int = TRUE,
  pval = TRUE,
  legend.labs = c("M", "F"),
  legend.title = "Sex",
  palette = c("cornflowerblue", "indianred3"),
  title = "Kaplan-Meier",
  xlab = "Time (days)")
```



5.2.9 Mosaic plot

Le mosaic plot permet de représenter le croisement de modalités de variables qualitatives. Il permet notamment de vérifier quels sont les croisements qui sont en sur ou sous effectifs par rapport à une situation où les deux variables seraient indépendantes.

```
library(vcd)
tab <- xtabs(~cut + color, diamonds)
mosaic(tab,
       shade = TRUE,
       legend = TRUE)
```



Exercice 4.2

- Sur les données **lung** utilisées précédemment, réaliser un “mosaic plot” entre les variables **status** et **sex**.
- Toujours sur les données **lung**, réaliser un graphique de type “ridge plot” sur la variable variable **age** en fonction des modalités de la variable **status**.
- Réaliser un graphique des corrélations sur les variables **ph.karno**, **pat.karno**, **meal.cal**, **wt.loss** du jeu de données **lung**.

5.3 Présenter des tableaux de résultats

Lorsqu'on présente un tableau de résultats (par exemple les résultats d'une régression) dans un document de type rapport ou présentation, il est préférable de ne pas insérer les sorties de **R** à l'état brut. D'une part, l'esthétique du format de sortie n'est pas agréable à lire et d'autre part, si on prend par exemple les sorties de la fonction **lm()**, il y a énormément de valeurs qu'on ne prend pas la peine de commenter en général.

Plutôt que de faire du copier/coller des chiffres qui nous intéressent depuis **R**, il existe un certain nombres de fonctions dans **R** qui retournent du code, par exemple **html** ou **LaTeX**, prêt à être insérer directement dans un document de type **html** ou **LaTeX**.

L'utilisation de ce type de fonctions est facilitée dans les fichiers **R** Markdown. En effet, on peut utiliser l'option **results = 'asis'** dans un chunk pour indiquer que le résultat retourné sera dans le format du document de sortie (soit **html** si le document final est en **html**, soit **LaTeX** si le document final est **pdf**).

5.3.1 La fonction `kable()`

Il s'agit d'une fonction incluse dans le package **knitr**, qui est le package permettant de compiler des fichiers Sweave ou **R** Markdown dans plusieurs formats de sortie. Cette fonction s'applique sur des objets de type **matrix** ou **data.frame**. Pour l'utiliser dans **R** Markdown, il suffit de faire :

```
# ``{r, results = 'asis'}
# knitr::kable(mtcars)
# ````
```

Ce qui produira le résultat suivant. Ici, on a ajouté l'option **echo = F** dans le chunk pour ne pas afficher le code **R** :

```
mpg
cyl
disp
hp
drat
wt
qsec
vs
am
gear
carb
Mazda RX4
21.0
6
160.0
110
3.90
2.620
16.46
0
1
4
```

4

Mazda RX4 Wag

21.0

6

160.0

110

3.90

2.875

17.02

0

1

4

4

Datsun 710

22.8

4

108.0

93

3.85

2.320

18.61

1

1

4

1

Hornet 4 Drive

21.4

6

258.0

110

3.08

3.215
19.44
1
0
3
1
Hornet Sportabout
18.7
8
360.0
175
3.15
3.440
17.02
0
0
3
2
Valiant
18.1
6
225.0
105
2.76
3.460
20.22
1
0
3
1
Duster 360

14.3
8
360.0
245
3.21
3.570
15.84
0
0
3
4
Merc 240D
24.4
4
146.7
62
3.69
3.190
20.00
1
0
4
2
Merc 230
22.8
4
140.8
95
3.92
3.150
22.90

1
0
4
2

Merc 280

19.2

6

167.6

123

3.92

3.440

18.30

1

0

4

4

Merc 280C

17.8

6

167.6

123

3.92

3.440

18.90

1

0

4

4

Merc 450SE

16.4

8

275.8

180

3.07

4.070

17.40

0

0

3

3

Merc 450SL

17.3

8

275.8

180

3.07

3.730

17.60

0

0

3

3

Merc 450SLC

15.2

8

275.8

180

3.07

3.780

18.00

0

0

3

3

Cadillac Fleetwood

10.4

8

472.0

205

2.93

5.250

17.98

0

0

3

4

Lincoln Continental

10.4

8

460.0

215

3.00

5.424

17.82

0

0

3

4

Chrysler Imperial

14.7

8

440.0

230

3.23
5.345
17.42
0
0
3
4
Fiat 128
32.4
4
78.7
66
4.08
2.200
19.47
1
1
4
1
Honda Civic
30.4
4
75.7
52
4.93
1.615
18.52
1
1
4
2

Toyota Corolla

33.9

4

71.1

65

4.22

1.835

19.90

1

1

4

1

Toyota Corona

21.5

4

120.1

97

3.70

2.465

20.01

1

0

3

1

Dodge Challenger

15.5

8

318.0

150

2.76

3.520

16.87
0
0
3
2
AMC Javelin
15.2
8
304.0
150
3.15
3.435
17.30
0
0
3
2
Camaro Z28
13.3
8
350.0
245
3.73
3.840
15.41
0
0
3
4
Pontiac Firebird
19.2

8
400.0
175
3.08
3.845
17.05
0
0
3
2
Fiat X1-9
27.3
4
79.0
66
4.08
1.935
18.90
1
1
4
1
Porsche 914-2
26.0
4
120.3
91
4.43
2.140
16.70
0

1
5
2
Lotus Europa
30.4
4
95.1
113
3.77
1.513
16.90
1
1
5
2
Ford Pantera L
15.8
8
351.0
264
4.22
3.170
14.50
0
1
5
4
Ferrari Dino
19.7
6
145.0

175

3.62

2.770

15.50

0

1

5

6

Maserati Bora

15.0

8

301.0

335

3.54

3.570

14.60

0

1

5

8

Volvo 142E

21.4

4

121.0

109

4.11

2.780

18.60

1

1

4

2

Il existe un certain nombre d'options dans la fonction `kable()` :

- **digits=** définit le nombre de chiffres décimales à afficher,
- **align=** ('r', right, 'l', left ou 'c', center) indique la position des valeurs à l'intérieur des cellules.
- **caption=** ajoute une légende.

Une amélioration de la fonction `kable()` est la fonction `tbl()` du package **kableExtra**. Celle ci permet de représenter les valeurs dans un tableau avec des couleurs et/ou des tailles de police différentes. Par ailleurs, en cherchant un peu plus loin, il est également possible d'insérer des images ou des petits graphiques statistiques dans les cellules du tableau (voir la page suivante pour plus d'informations lien.)

On représente ici les 5 premières lignes du jeu de données **iris**:

```
library(kableExtra)
vs_dt <- iris[1:5, ]
vs_dt[1:4] <- lapply(vs_dt[1:4], function(x) {
  cell_spec(x, bold = T, color = spec_color(x, end = 0.9),
             font_size = spec_font_size(x))
})
vs_dt[5] <- cell_spec(vs_dt[[5]], color = "white", bold = T,
                      background = spec_color(1:5, end = 0.9, option = "A", direction = -1))
kbl(vs_dt, escape = F, align = "c") %>% kable_classic("striped", full_width = F)
```

5.3.2 La fonction `stargazer()`

La fonction `stargazer()` du package **stargazer** est un plus général que la fonction `kable()` car elle prend en compte non seulement des objets de type **matrix** et **data.frame**, mais aussi **lm**.

On l'utilise ainsi dans un document **R** Markdown :

```
# `~{r, results = 'asis'}
# stargazer::stargazer(attitude)
#`~`
```

5.3.2.1 Table de données brutes

Pour afficher une table de données brutes, on utilise la fonction `stargazer()`. Parmi les nombreuses options de cette fonction, on en cite ici quelques-unes :

- **summary=F** signifie que la table doit être représentée à l'état brut (par défaut, la fonction appliquée à un **data.frame** va faire un résumé statistique du **data.frame**),

- **digits=1** indique qu'un seul chiffre après la virgule sera représenté,
- **type=** indique le format dans lequel la table sera retournée, par exemple “**latex**” (par défaut) ou “**html**”.
- **header=F** précise de ne pas ajouter la date, l'heure, etc. à laquelle la table a été créée,
- **rownames=F** précise de ne pas afficher le nom des lignes,
- **title=** permet de donner un titre à la table,

Dans **R** Markdown, on a ajouté dans le chunk l'option **results = 'asis'** pour indiquer que **R** allait ressortir un résultat qui est dans le format du support de sortie utilisé (que ce soit *html* ou *latex*)

```
stargazer::stargazer(head(iris), summary = F, digits = 1, type = "html",
                      header = F, rownames = F, title = "Tableau de données"
)
```

Tableau de données

Sepal.Length

Sepal.Width

Petal.Length

Petal.Width

Species

binary

5.1

3.5

1.4

0.2

setosa

1

4.9

3

1.4

0.2

setosa

1

```
4.7  
3.2  
1.3  
0.2  
setosa  
1  
4.6  
3.1  
1.5  
0.2  
setosa  
1  
5  
3.6  
1.4  
0.2  
setosa  
1  
5.4  
3.9  
1.7  
0.4  
setosa  
1
```

5.3.2.2 Tableau de résumé

Par défaut, la fonction va créer un tableau de résumé statistique :

```
stargazer::stargazer(iris, type = "html", nobs = FALSE, mean.sd = TRUE,  
                      median = TRUE, iqr = TRUE, header = F, title = "Résumé",  
                      digit.separate = c(3, 3, 1, 1, 1, 1, 1))
```

Résumé

Statistic

Mean
St. Dev.
Min
Pctl(25)
Median
Pctl(75)
Max
Sepal.Length
5.843
0.828
4.300
5.100
5.800
6.400
7.900
Sepal.Width
3.057
0.436
2.000
2.800
3.000
3.300
4.400
Petal.Length
3.758
1.765
1.000
1.600
4.350
5.100
6.900

```
Petal.Width
1.199
0.762
0.100
0.300
1.300
1.800
2.500
```

5.3.2.3 Tableaux de régression

Un des gros avantages de la fonction `stargazer()` est qu'elle permet de synthétiser des modèles de régression.

```
output_1 <- lm(Sepal.Length ~ Species, data = iris)
output_2 <- lm(Sepal.Length ~ Sepal.Width, data = iris)

stargazer::stargazer(output_1, output_2, type = "html",
                      title = "Résultat de régression", header = F)
```

Résultat de régression

Dependent variable:

Sepal.Length

(1)

(2)

Speciesversicolor

0.930***

(0.103)

Speciesvirginica

1.582***

(0.103)

Sepal.Width

-0.223

(0.155)

Constant

5.006***

6.526***

(0.073)

(0.479)

Observations

150

150

R2

0.619

0.014

Adjusted R2

0.614

0.007

Residual Std. Error

0.515 (df = 147)

0.825 (df = 148)

F Statistic

119.265*** (df = 2; 147)

2.074 (df = 1; 148)

Note:

p<0.1; p<0.05; p<0.01

Elle permet également de comparer des modèles entre eux. Pour en savoir plus, on pourra consulter le document suivant :

- <https://www.jakeruss.com/cheatsheets/stargazer/#html-formatting>

Exercice 4.3

Insérer dans un document Markdown :

- le tableau de corrélation des variables quantitatives du jeu de données **iris**.
- le tableau de régression du jeu de données **iris**

5.4 Graphiques interactifs

5.4.1 Via plotly

```
require("plotly")
```

Ici, nous allons présenter quelques exemples d'utilisations du package **plotly** qui permet de faire des graphiques interactifs en utilisant une interface web via la bibliothèque `plotly.js` écrite en JavaScript.

Il ne s'agit ici que d'une brève introduction, le lecteur pourra consulter le lien suivant pour plus d'informations :

- <https://plot.ly/r/>

Lorsqu'on utilise les fonctions de ce package depuis **RStudio**, la fenêtre graphique qui s'ouvre depuis **RStudio** rend possible l'interactivité. Cela a pour effet de créer un graphique dans lequel il est possible d'obtenir des informations supplémentaires lorsqu'on place le curseur sur des zones de celui-ci.

De plus, lorsqu'on intègre ce type de graphique dans un document **R** Markdown, cela garde l'interactivité dans la page html créée.

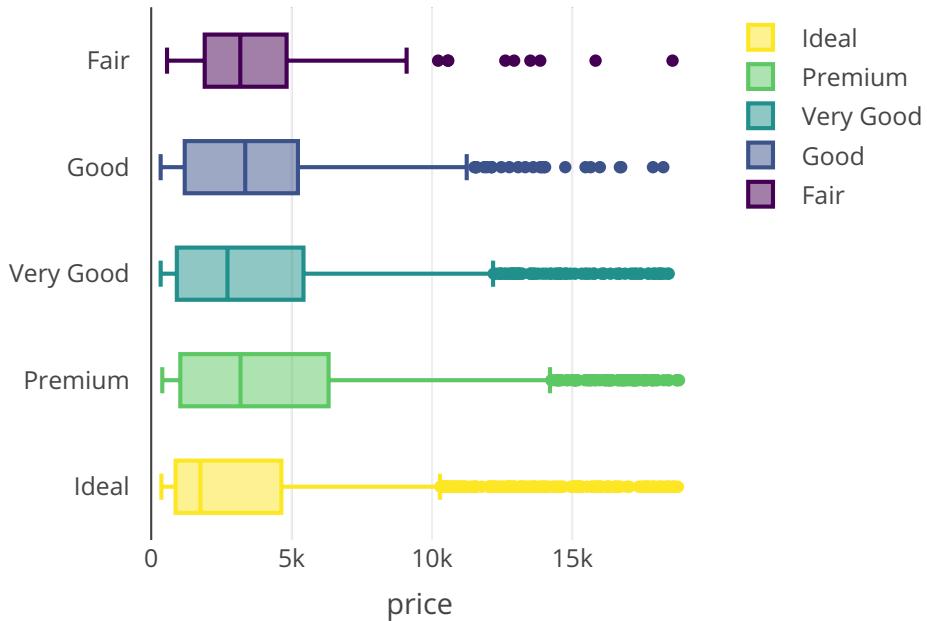
Son utilisation est la suivante : on appelle la fonction `plot_ly()` dans lequel :

- le 1er argument est le nom du jeu de données,
- l'argument `x = ~` donne le nom de la variable d'intérêt,
- l'argument `color = ~` donne le nom d'une variable conditionnelle,
- l'argument `type =` donne le type de graphique qu'on souhaite représenter.

Par exemple, pour une boîte à moustache, on utilise la syntaxe suivante :

```
p <- plot_ly(diam_ech, x = ~price, color = ~cut, type = "box")
```

```
p
```

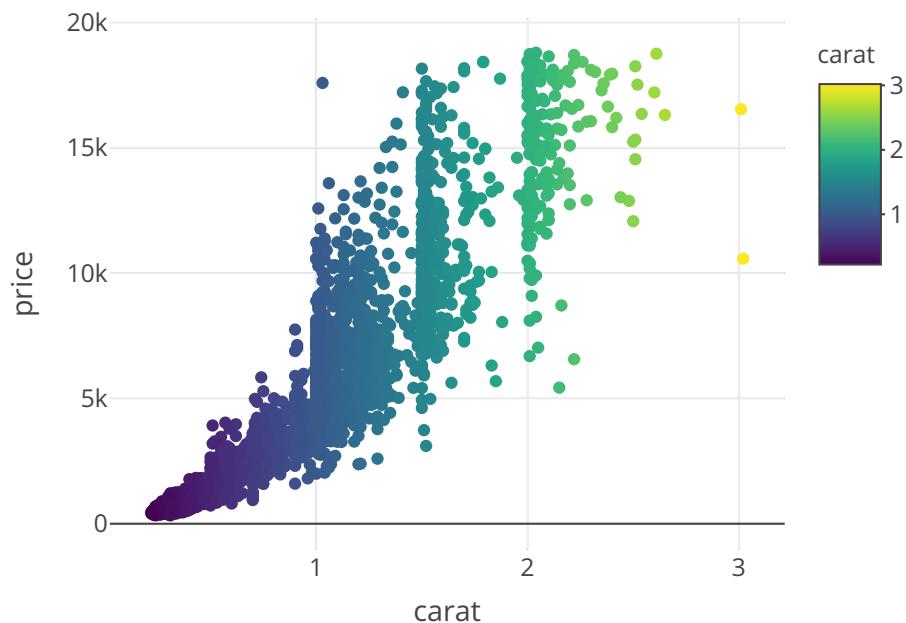


;

Pour un nuage de points, on utilise la syntaxe suivante :

```
p <- plot_ly(diam_ech, x = ~carat, y = ~price, type = "scatter", mode = "markers",
              hoverinfo = 'text',
              text = ~paste('Carat: ', carat,
                           '\n Price: ', price,
                           '\n Clarity: ', diam_ech$clarity),
              color = ~carat)
```

p

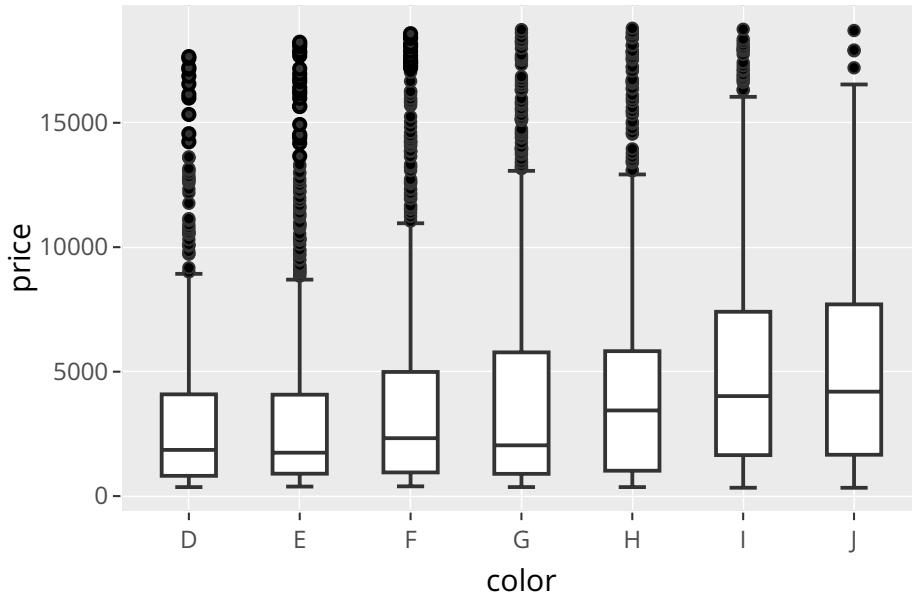


;

On peut si on le souhaite coupler les graphiques **ggplot2** avec l'interactivité de **plotly**. Pour cela, on peut reprendre n'importe lequel des graphiques présentés dans la section précédente et utiliser la fonction *ggplotly()* pour rendre possible l'interaction :

```
p <- ggplot(diam_ech) +
  aes(x = color, y = price) +
  geom_boxplot()

ggplotly(p)
```



;

Autres packages permettant de faire des graphiques interactifs :

- **ggvis** : voir par exemple <https://ggvis.rstudio.com/>
- **rCharts** voir par exemple <https://ramnathv.github.io/rCharts/>

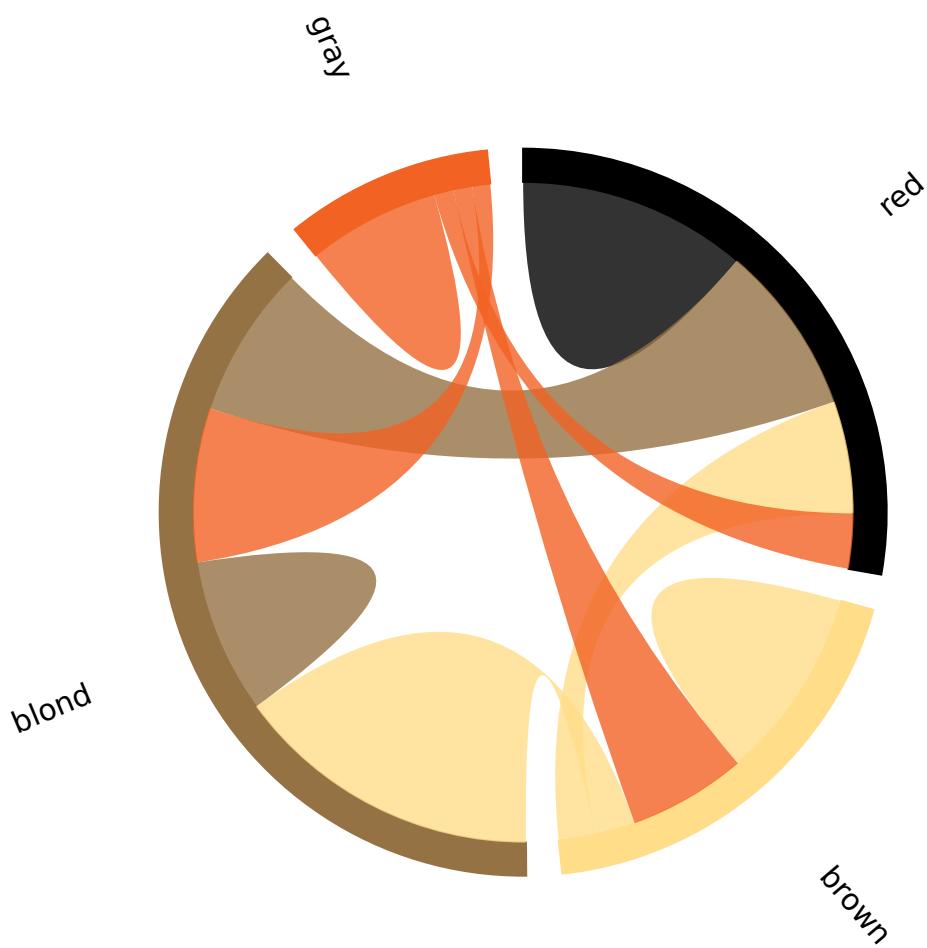
5.4.2 Via D3

“D3.js (ou D3 pour Data-Driven Documents) est une bibliothèque graphique **JavaScript** qui permet l'affichage de données numériques sous une forme graphique et dynamique. Il s'agit d'un outil important pour la conformation aux normes **W3C** qui utilise les technologies courantes **SVG**, **JavaScript** et **CSS** pour la visualisation de données” (définition donnée par Wikipedia).

Il existe un certain nombre de packages **R** qui permettent de créer des graphiques en D3. Parmi ces packages, on nommera **r2d3** (voir <https://rstudio.github.io/r2d3/> pour plus d'informations).

On présente ici un exemple provenant d'un autre package (**networkD3**), qui permet de représenter des données de flux.

```
library(networkD3)
hairColourData <- matrix(c(11975, 1951, 8010, 1013, 5871, 10048, 16145, 990,
                           8916, 2060, 8090, 940, 2868, 6171, 8045, 6907),
                           nrow = 4)
chordNetwork(Data = hairColourData, width = 500, height = 500,
             colourScale = c("#000000", "#FFDD89", "#957244", "#F26223"),
             labels = c("red", "brown", "blond", "gray"))
```



5.5 Mini-introduction à shiny

shiny est un package développé par **RStudio** qui permet la création de pages web interactives, sans avoir à connaître ni HTML, ni CSS, ni JavaScript.

On renvoie le lecteur vers la page web **RStudio** (<http://shiny.rstudio.com/gallery/>) pour voir des exemples d'applications réalisées avec **shiny**.

Pour utiliser **shiny**, cela se fait de manière extrêmement intuitive depuis **RStudio**.

Cette section s'inspire fortement de la présentation suivante :

- https://www.londonr.org/wp-content/uploads/sites/2/presentation_s/LondonR_-_Workshop-Introduction_to_Shiny_-_Aimee_Gott_-

_20150330.pdf

Le lecteur pourra également consulter cette présentation de Christophe Bon-temps (TSE) :

- <https://vimeo.com/301596705>

5.5.1 Un premier exemple

Dans un premier temps, on commence par créer deux fichiers qu'il faudra enregistrer dans un même répertoire.

- le premier fichier, en général nommé “ui.R”, va contenir le code qui permet de paramétrer la disposition des différents éléments que l'on souhaite afficher dans la page web. Dans l'exemple par défaut, on découpe la page web en deux parties : une partie à gauche qui contient des éléments informatifs (un titre, etc.) ainsi que des paramètres que l'utilisateur pourra éventuellement modifier. La partie à droite contiendra un graphique que l'on souhaite représenter.

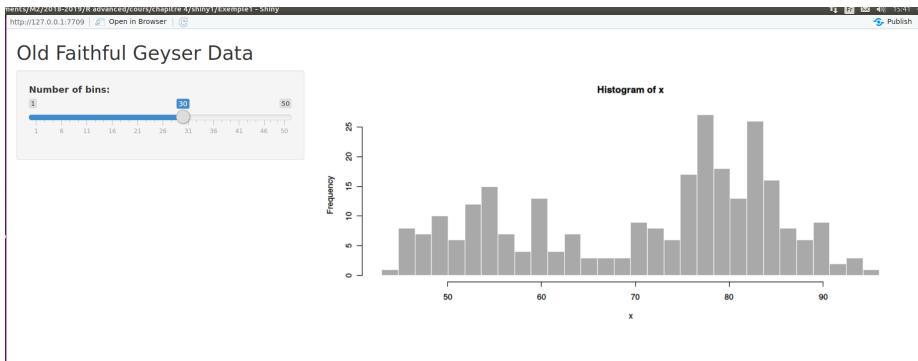


Figure 5.1: Screenshot de shiny

- le second fichier, en général nommé “server.R” contient le code qui va permettre de constituer le graphique que l'on souhaite représenter dans la partie à droite de l'application. Pour construire ce graphique, on utilisera des paramètres qui ont été définis par l'utilisateur depuis l'interface.

L'idée est donc que les deux fichiers dépendent l'un de l'autre. En effet, dès que l'utilisateur va modifier des paramètres sur l'interface, cela va modifier des objets dans le fichier “ui.R” ce qui aura pour conséquence d'appeler le fichier “server.R” afin de représenter le nouveau graphique qui aura pris en compte la modification effectuée par l'utilisateur.

On va commencer par un exemple simple. Pour cela, depuis **RStudio**, aller dans “File”, puis “New File”, puis “Shiny web app”. Donner un nom à votre

api, laisser les options par défaut et choisir le répertoire dans lequel vous aller sauvegarder vos deux fichiers.

Le fichier “ui.R” créé par défaut est celui-ci. Nous avons ajouté ici les commentaires pour expliquer les différentes étapes de création de la page web :

```
library("shiny") # appel de la librairie "shiny"

shinyUI( # Création d'une application
  fluidPage( # mise en page shiny standard qui permet d'adapter l'interface
    # au navigateur utilisé

    titlePanel("Old Faithful Geyser Data"), # titre de l'appli

    sidebarLayout( # fonction qui spécifie quelles sont les différentes parties
      # de la page web. Ici, il y en aura 2 :
      # - une partie à gauche (sidebarPanel)
      # - une partie à droite (mainPanel)

      sidebarPanel( # - la partie à gauche sera une zone grisée qui
        # peut contenir différentes choses (texte, réglette, etc.)
        # Ici, il y aura une réglette (fonction sliderInput()):
        sliderInput("bins", # le 1er argument est le nom du paramètre dont
          # la valeur vaut ce que l'utilisateur choisit
          # depuis l'interface. Cette valeur pourra être
          # utilisée depuis le fichier "server.R"
          "Number of bins:", # le titre de la réglette
          min = 1,           # la valeur minimum de la réglette
          max = 50,          # la valeur maximum de la réglette
          value = 30)       # la valeur par défaut de "bins"
    ),

    mainPanel( # la partie à droite est la zone principale de la page web
      plotOutput("distPlot") # elle contiendra un graphique dont le nom est
                            # distPlot et qui sera défini depuis le fichier
                            # "server.R"
    )
  )
)
```

Le fichier “server.R” créé par défaut est celui-ci :

```
library("shiny") # appel de la librairie "shiny"

shinyServer( # ouvre la connection web et va autoriser le partage d'infos
  # entre les fichiers "ui.R" et "server.R"
  function(input, output) { # - input : contient l'environnement produit par
```

```

# "ui.R"
# - output contient l'environnement produit par
# "server.R"
output$distPlot <- renderPlot({ # on crée dans l'environnement output un
                                # graphique appelé distPlot

  x      <- faithful[, 2] # données à représenter
  # on crée les classes de l'histogramme en utilisant l'information bins
  # provenant de l'environnement "input"
  bins <- seq(min(x), max(x), length.out = input$bins + 1)

  # on représente l'histogramme
  hist(x, breaks = bins, col = 'darkgray', border = 'white')
}

})

```

Depuis **RStudio**, lorsque vous avez ouvert l'un des deux fichiers "ui.R" ou "server.R" il est possible de lancer l'application en cliquant sur le bouton "Run App". Cela a pour effet d'ouvrir une fenêtre *html* depuis **RStudio** où apparaît d'une part sur la gauche l'interface (codée dans le fichier "ui.R") et sur la droite le graphique (codé dans le fichier "server.R"). Dans cet exemple, lorsqu'on bouge le curseur de la réglette, cela a pour effet de modifier le graphique automatiquement.

Remarque : tant que vous n'aurez pas fermé la fenêtre *html*, vous n'aurez pas la main sur la console **R** de **RStudio**.

5.5.2 Quelques fonctionnalités de shiny

5.5.2.1 Dans la partie de gauche

Dans l'exemple ci-dessus, nous avons seulement inséré une réglette dans la partie *sidebarPanel()*. On aurait pu ajouter d'autres types d'outils très intéressants en utilisant une des fonctions ci-dessous :

- *textInput()* : saisie d'une chaîne de caractères,
- *numericInput()* : saisie d'une valeur numérique,
- *selectInput()* : choix parmi une ou plusieurs valeurs sous forme de liste déroulante,
- *sliderInput()* : réglette de valeurs numériques,
- *radioButtons()* : choix de valeurs avec des boutons,
- *fileInput()* : choisir un fichier (en général de données).

5.5.2.2 Dans la partie de droite

Dans l'exemple ci-dessus, nous avons affiché dans la partie de droite seulement un graphique. Pour cela nous avons utilisé la fonction *renderPlot()* depuis le fichier “server.R” et *plotOutput()* depuis le fichier “ui.R”. On peut également affiché :

- du texte : on utilisera la fonction *renderPrint()* depuis le fichier “server.R” et *textOutput()* depuis le fichier “ui.R”
- des tables de données : on utilisera la fonction *renderDataTable()* depuis le fichier “server.R” et *dataTableOutput()* depuis le fichier “ui.R”
- des images : on utilisera la fonction *renderImage()* depuis le fichier “server.R” et *imageOutput()* depuis le fichier “ui.R”

5.5.3 Publier ses applications

Une fois vos applications développées en local, il est possible de les rendre accessible à tous grâce aux serveurs de chez **RStudio** qui propose des offres d'hébergement. La première offre est gratuite au-dessous d'un certain nombre d'applications et temps d'utilisation. Pour plus d'informations, consulter : <https://www.rstudio.com/products/shiny/shiny-server/>

Générer des documents de sortie : <https://stackoverflow.com/questions/66237085/generating-downloadable-reports-from-shiny-app>

Chapter 6

Hello bookdown

All chapters start with a first-level heading followed by your chapter title, like the line above. There should be only one first-level heading (#) per .Rmd file.

6.1 A section

All chapter sections start with a second-level (##) or higher heading followed by your section title, like the sections above and below here. You can have as many as you want within a chapter.

An unnumbered section

Chapters and sections are numbered by default. To un-number a heading, add a {.unnumbered} or the shorter {-} at the end of the heading, like in this section.

Chapter 7

Cross-references

Cross-references make it easier for your readers to find and link to elements in your book.

7.1 Chapters and sub-chapters

There are two steps to cross-reference any heading:

1. Label the heading: `# Hello world {#nice-label}`.
 - Leave the label off if you like the automated heading generated based on your heading title: for example, `# Hello world = # Hello world {#hello-world}`.
 - To label an un-numbered heading, use: `# Hello world {-#nice-label}` or `{# Hello world .unnumbered}`.
2. Next, reference the labeled heading anywhere in the text using `\@ref(nice-label)`; for example, please see Chapter 7.
 - If you prefer text as the link instead of a numbered reference use: any text you want can go here.

7.2 Captioned figures and tables

Figures and tables *with captions* can also be cross-referenced from elsewhere in your book using `\@ref(fig:chunk-label)` and `\@ref(tab:chunk-label)`, respectively.

See Figure 7.1.

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

Don't miss Table 7.1.

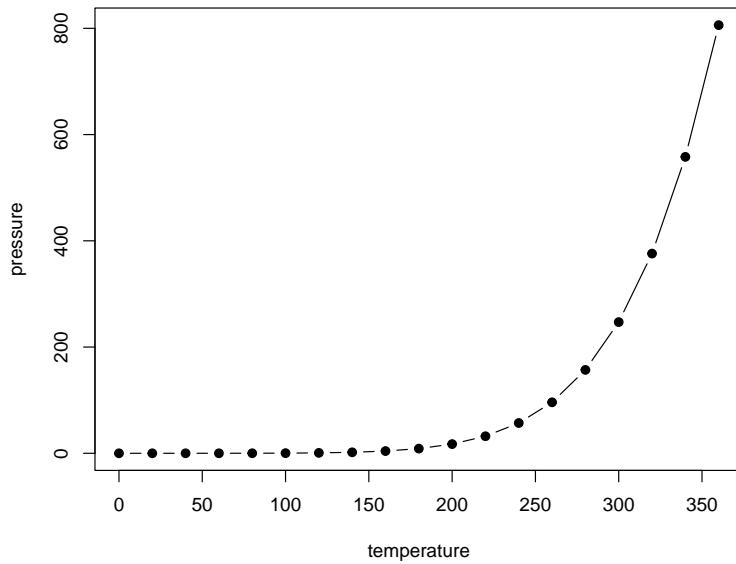


Figure 7.1: Here is a nice figure!

```
knitr::kable(  
  head(pressure, 10), caption = 'Here is a nice table!',  
  booktabs = TRUE  
)
```

Table 7.1: Here is a nice table!

| temperature | pressure |
|-------------|----------|
| 0 | 0.0002 |
| 20 | 0.0012 |
| 40 | 0.0060 |
| 60 | 0.0300 |
| 80 | 0.0900 |
| 100 | 0.2700 |
| 120 | 0.7500 |
| 140 | 1.8500 |
| 160 | 4.2000 |
| 180 | 8.8000 |

Chapter 8

Parts

You can add parts to organize one or more book chapters together. Parts can be inserted at the top of an .Rmd file, before the first-level chapter heading in that same file.

Add a numbered part: `# (PART) Act one {-} (followed by # A chapter)`

Add an unnumbered part: `# (PART*) Act one {-} (followed by # A chapter)`

Add an appendix as a special kind of un-numbered part: `# (APPENDIX) Other stuff {-} (followed by # A chapter)`. Chapters in an appendix are prepended with letters instead of numbers.

Chapter 9

Footnotes and citations

9.1 Footnotes

Footnotes are put inside the square brackets after a caret `^[]`. Like this one¹.

9.2 Citations

Reference items in your bibliography file(s) using `@key`.

For example, we are using the **bookdown** package [Xie, 2021] (check out the last code chunk in index.Rmd to see how this citation key was added) in this sample book, which was built on top of R Markdown and **knitr** [Xie, 2015] (this citation was added manually in an external file book.bib). Note that the `.bib` files need to be listed in the index.Rmd with the YAML `bibliography` key.

The RStudio Visual Markdown Editor can also make it easier to insert citations:
<https://rstudio.github.io/visual-markdown-editing/#/citations>

¹This is a footnote.

Chapter 10

Blocks

10.1 Equations

Here is an equation.

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k} \quad (10.1)$$

You may refer to using `\@ref(eq:binom)`, like see Equation (10.1).

10.2 Theorems and proofs

Labeled theorems can be referenced in text using `\@ref(thm:tri)`, for example, check out this smart theorem 10.1.

Theorem 10.1. *For a right triangle, if c denotes the length of the hypotenuse and a and b denote the lengths of the **other** two sides, we have*

$$a^2 + b^2 = c^2$$

Read more here <https://bookdown.org/yihui/bookdown/markdown-extensions-by-bookdown.html>.

10.3 Callout blocks

The R Markdown Cookbook provides more help on how to use custom blocks to design your own callouts: <https://bookdown.org/yihui/rmarkdown-cookbook/custom-blocks.html>

Chapter 11

Sharing your book

11.1 Publishing

HTML books can be published online, see: <https://bookdown.org/yihui/bookdown/publishing.html>

11.2 404 pages

By default, users will be directed to a 404 page if they try to access a webpage that cannot be found. If you'd like to customize your 404 page instead of using the default, you may add either a `_404.Rmd` or `_404.md` file to your project root and use code and/or Markdown syntax.

11.3 Metadata for sharing

Bookdown HTML books will provide HTML metadata for social sharing on platforms like Twitter, Facebook, and LinkedIn, using information you provide in the `index.Rmd` YAML. To setup, set the `url` for your book and the path to your `cover-image` file. Your book's `title` and `description` are also used.

This `gitbook` uses the same social sharing data across all chapters in your book—all links shared will look the same.

Specify your book's source repository on GitHub using the `edit` key under the configuration options in the `_output.yml` file, which allows users to suggest an edit by linking to a chapter's source file.

Read more about the features of this output format here:

<https://pkgs.rstudio.com/bookdown/reference/gitbook.html>

Or use:

```
?bookdown::gitbook
```

Bibliography

Yihui Xie. *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition, 2015. URL <http://yihui.org/knitr/>. ISBN 978-1498716963.

Yihui Xie. *bookdown: Authoring Books and Technical Documents with R Markdown*, 2021. URL <https://CRAN.R-project.org/package=bookdown>. R package version 0.24.