

Guide R

Prof. Audrey Bürki, Samuel Arthers, Mégane Bollenrücher

2025-02-17

Contents

1	Introduction	5
2	Installation et environnement R et Rstudio	7
2.1	Présentation des logiciels	7
2.2	Installation	7
2.3	Environnement de travail	8
3	Objets et opérateurs	11
3.1	Objets dans R	11
3.2	Opérateurs logiques	19
4	Packages et données	21
4.1	Installation et gestion des packages	21
4.2	Téléchargement des données	21
5	Description des données quantitatives et qualitatives	25
5.1	Description numérique	25
5.2	Description graphique	28
6	Distribution de probabilités	33
6.1	Distribution normale	33
6.2	Autres distributions	36
7	Les boucles FOR	37
7.1	Structure	37
7.2	Utilisation	37

Merci de prendre note que ce bookdown est en cours de rédaction.

Chapter 1

Introduction

Ceci est le guide R que nous proposons pour vous accompagner durant les travaux pratiques de méthodologique.

Chapter 2

Installation et environnement R et Rstudio

2.1 Présentation des logiciels

R est un langage de programmation adapté au traitement de données et à l'analyse statistique.

Pour programmer en langage R, il est nécessaire d'installer deux outils essentiels:

1. Le **logiciel R** permet de traduire du texte sous forme de code R en binaire qui est le langage interne du processeur de l'ordinateur.
2. Le **logiciel RStudio** permet de faciliter l'utilisation du logiciel R en donnant l'accès à une interface utilisateur.

Il est possible de faire une analogie avec une voiture. Le logiciel R est le moteur et RStudio est le tableau de bord. Sans le tableau de bord, il n'est pas possible de contrôler le moteur.

2.2 Installation

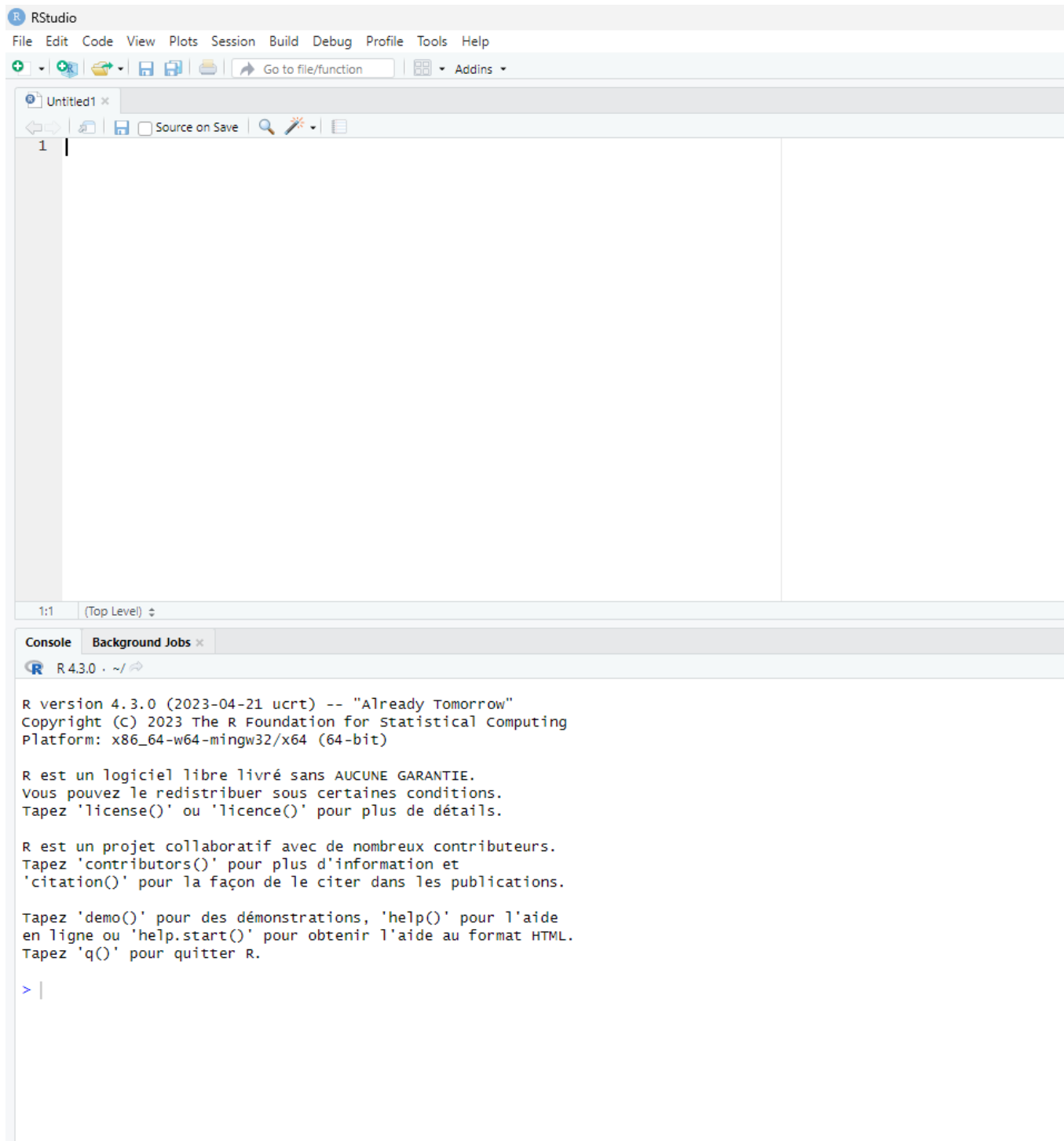
1. Installer R sur le site de R.
 - i. Choisir et télécharger la version de R selon votre système d'exploitation.
 - Pour windows : <https://cran.r-project.org/bin/windows/base/>
 - Pour MAC : <https://cran.r-project.org/bin/macosx/>
 - Pour Linux : <https://cran.r-project.org/index.html>

- ii. Installer le logiciel R sur votre ordinateur en exécutant le fichier téléchargé.
2. Installer RStudio sur le site suivant: <https://posit.co/download/rstudio-desktop/>

Après avoir installé ces deux logiciels, vous aurez accès à deux nouvelles applications. Cependant, nous utiliserons uniquement RStudio pour programmer. Lorsque vous exécuterez votre code écrit sur RStudio, ce dernier fera automatiquement appel à R pour exécuter les codes.

2.3 Environnement de travail

Une fois que RStudio est lancé, une interface découpée en plusieurs zones se présente. Ces parties parties peuvent être redimensionnées, masquées ou maximisées selon vos préférences.



Chacune des quatre zones a sa propre utilité:

1. Cette zone est dédiée aux fichiers sources. Ce volet permet d'écrire et de sauvegarder les lignes de code. Ce sera la partie la plus utilisée lors de la programmation. Un nouveau script peut être ouvert à partir de l'onglet **File** en haut à gauche de l'écran, puis **New File** et **R Script**. Chaque script peut être enregistré soit via le même onglet en choisissant **Save As**, soit en utilisant les raccourcis classiques de votre clavier.
2. Cette zone fournit des informations sur les objets, les variables et les données en mémoire sous l'onglet **Environment**.
3. La console est affichée en bas à gauche. Cette partie permet d'entrer et d'exécuter des instructions et voir les résultats s'afficher.
4. Cette zone permet de naviguer dans le répertoire de travail dans l'onglet **Files**, d'afficher les graphes réalisés dans l'onglet **Plots**, d'afficher les extensions/packages disponibles sous l'onglet **Packages** et également d'afficher l'aide (qui est très complète) sous l'onglet **Help**.

Chapter 3

Objets et opérateurs

3.1 Objets dans R

Une variable permet de stocker une valeur ou un objet dans R. De cette façon, il sera possible d'accéder à la valeur ou à l'objet qui est stocké dans la variable.

```
# Assignment de la valeur 3 à la variable "ma_variable"  
ma_variable <- 3
```

La ligne de code ci-dessus déclare une variable nommée “ma_variable” et lui assigne la valeur de 3. L'exécution de ce code n'affiche pas de résultat dans la console, mais l'objet nommé “ma_variable” est bien créée et stockée dans l'environnement.

Pour afficher le contenu de la variable, il suffit de taper le nom de celle-ci pour l'afficher dans la console.

```
# Affichage du contenu de la variable "ma_variable"  
ma_variable
```

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

De plus, il est également possible d'utiliser la fonction `print()` qui permet d'afficher la valeur ou l'objet de la variable sélectionnée.

```
# Affichage du contenu de la variable "ma_variable"  
print(ma_variable)
```

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

En langage R, il existe différents types d'objets qui peuvent être assignés à des variables comme les scalaires, les vecteurs, les facteurs, les matrices et les bases de données. Ces objets sont présentés dans les points suivants.

3.1.1 Scalaire

Un scalaire permet de stocker un objet sous forme de valeur numérique, de chaîne de caractères ou de valeur logique.

```
# Scalaire numerique
a <- 3
a
```

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

Une chaîne de caractères est une suite de caractères qui doit être écrit entre guillemets (" ").

```
# Scalaire sous forme d'une chaine de caractères
b <- "Statistique"
b
```

```
## [1] "Statistique"
```

Une valeur logique est une quantité binaire (vrai ou faux). Ces variables s'écrivent TRUE et FALSE. Il est également possible d'utiliser T et F comme abréviations.

```
# Scalaire logique
c <- TRUE
c
```

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

```
d <- F
d
```

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

3.1.2 Vecteur

Un vecteur est un objet qui permet de stocker une liste ordonnée d'éléments. Les éléments d'un vecteur doivent être du même type. Pour pouvoir stocker une information dans un seul objet, il faut utiliser la fonction `c()` qui permet de combiner les arguments de la fonction.

```
# Vecteur numérique
a <- c(1, 2, 3, 4, 5, 6)
a
```

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

```
# Vecteur sous forme d'une chaine de caractères
b <- c("Un", "Deux", "Trois", "Quatre", "Cinq", "Six")
b
```

```
## [1] "Un"      "Deux"    "Trois"   "Quatre"  "Cinq"    "Six"
```

```
# Vecteur logique
c <- c(TRUE, FALSE, TRUE, FALSE, FALSE, TRUE)
c
```

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

Étant donné que le vecteur est un objet ordonné, il est possible d'accéder, remplacer ou modifier un ou plusieurs éléments par rapport à leur position dans l'objet. Il est nécessaire d'utiliser les crochets [] pour indiquer le ou les éléments à manipuler. Afin de connaître la longueur d'un vecteur, il faut utiliser la fonction `length()`.

```
# Vecteur logique
length(b)
```

```
## [1] 6
```

```
length(c)
```

```
## [1] 6
```

Pour accéder un seul élément du vecteur, il suffit d'écrire le nom de la variable suivi de crochets contenant la position de l'élément sélectionné.

```
# Extraction d'un élément:
b[2]
```

```
## [1] "Deux"
```

Il est possible d'accéder à plusieurs éléments en mettant un vecteur de position entre crochets.

```
# Extraction de plusieurs éléments:
b[c(2,3,5)]
```

```
## [1] "Deux" "Trois" "Cinq"
```

Il est aussi possible d'accéder à une série d'éléments à la suite en mettant le signe : entre les 2 positions désirées.

```
# Extraction d'une série d'éléments:
b[2:5]
```

```
## [1] "Deux" "Trois" "Quatre" "Cinq"
```

Il est également possible d'accéder à certaines lignes en fonction de la valeur logique d'un vecteur ayant la même taille du vecteur sélectionné. Si la position est mise à TRUE, la valeur sera sélectionnée et dans le cas dans lequel la valeur est mise à FALSE la valeur ne sera pas retenue.

```
# Extraction de plusieurs éléments en fonction de la valeur logique:
b[c(FALSE, TRUE, TRUE, FALSE, TRUE, TRUE)]
```

```
## [1] "Deux" "Trois" "Cinq" "Six"
```

Il est également possible de créer un vecteur vide dans lequel il sera par la suite possible d'ajouter des valeurs. Le code qui suit permet de créer ce vecteur:

```
vide <- vector()
```

3.1.3 Facteur

Un facteur est un vecteur dont les éléments peuvent prendre que des valeurs prédéfinies. Un facteur dispose de l'argument `levels` qui permet de définir des catégories de valeurs. Le facteur est généralement utilisé pour stocker des variables catégorielles.

Pour commencer, il faut définir un vecteur qui peut être numérique, logique ou chaîne de caractères. Le facteur est une variable nominale.

```
genre <- c("Homme", "Femme", "Femme", "Femme", "Homme")
genre
```

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

La fonction `factor()` permet de créer un facteur à partir d'un vecteur.

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

```
## [1] Homme Femme Femme Femme Homme
## Levels: Femme Homme
```

On note que la sortie est légèrement différente lorsque le vecteur est mis sous forme de facteur à 2 niveaux (Femme, Homme). Ceci s'affiche à la ligne `Levels`. Par défaut, les niveaux d'un facteur sont affichés par ordre alphabétique et numérique croissant. Il est possible de fixer l'ordre en ajoutant l'argument `levels` en appliquant la fonction `factor()`.

```
sexe <- c("H", "F", "F", "F", "H")
sexe <- factor(sexe, levels = c("H", "F"))
sexe
```

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

Dans le cas dans lequel on aimerait modifier un élément, il n'est pas possible d'affecter une valeur qui n'est pas défini comme un niveau. On voit donc apparaître une erreur dans la console.

```
genre[2] <- "Fille"
```

```
## Warning in `[<-factor`(`*tmp*`, 2, value = "Fille"): niveau de facteur
## incorrect, NAs générés
```

Il est possible de renommer les niveaux en utilisant la fonction `levels()`. Il faut faire attention à l'ordre lorsqu'on utilise la fonction `levels()`. L'argument `order` permet d'ordonner les labels proposés. Le facteur est dès lors une variable ordinale.

```
levels(genre) <- c("Fille", "Garcon")
genre
```

```
## [1] Garcon <NA>   Fille  Fille  Garcon
## Levels: Fille Garcon
```

Il est possible d'avoir un facteur numérique en y affectant une catégorie avec l'argument `labels` lors de l'utilisation de la fonction `factor()`.

```
satisfaction <- factor(c(3, 3, 4, 1, 2, 1, 1),
                        labels = c("Pas du tout d'accord", "Pas d'accord",
                                   "D'accord", "Tout à fait d'accord"),
                        order = TRUE)
satisfaction
```

```
## [1] D'accord          D'accord          Tout à fait d'accord
## [4] Pas du tout d'accord Pas d'accord      Pas du tout d'accord
## [7] Pas du tout d'accord
## 4 Levels: Pas du tout d'accord < Pas d'accord < ... < Tout à fait d'accord
```

3.1.4 Matrice

Une matrice est un vecteur dont les éléments sont disposés sous forme d'un tableau qui comporte des lignes et des colonnes. De façon équivalente au vecteur, les éléments de la matrices doivent être de même classe (numérique, logique ou chaîne de caractères). La fonction `matrix()` permet de déclarer une matrice. Il faut ajouter l'argument `ncol` et/ou `nrow` pour déterminer la forme de la matrice.

```
A <- matrix(1:24, nrow=6, ncol=4, byrow=FALSE)
A
```

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

Par défaut, le remplissage se fait par colonne. Il faut donc mettre l'argument `byrow` à `TRUE` pour remplir la matrice par ligne.

```
B <- matrix(1:24, nrow=6, ncol=4, byrow=TRUE)
B
```

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

L'objet matrice dispose de la fonction `dim()` qui permet d'obtenir sa dimension. Le premier terme correspond aux nombres de lignes et le deuxième correspond aux nombres de colonnes.

```
dim(B)
```

```
## [1] 6 4
```

Les fonctions `rownames()` et `colnames()` permettent de récupérer ou de définir les noms des lignes et des colonnes. Attention de bien mettre le bon nombre de noms aux lignes et aux colonnes.

```
rownames(B) <- c("L1", "L2", "L3", "L4", "L5", "L6")
colnames(B) <- c("C1", "C2", "C3", "C4")
B
```

```
##      C1 C2 C3 C4
## L1   1  2  3  4
## L2   5  6  7  8
## L3   9 10 11 12
## L4  13 14 15 16
## L5  17 18 19 20
## L6  21 22 23 24
```

Comme pour le vecteur, il est possible d'accéder à un ou plusieurs éléments de la matrice. Pour extraire une ligne de la matrice, il faut utiliser les crochets avec une virgule pour délimiter les deux dimensions de la matrice `[,]`. Le premier terme (celui avant la virgule) permet d'accéder aux colonnes et le deuxième terme (celui après la virgule) permet d'accéder aux lignes. Pour accéder à un seul élément, il faut indiquer la position de la ligne et de la colonne désirée.

```
# Extraction d'un seul élément
A[2,3]
```

```
## [1] 14
```

Pour accéder à une ligne complète, il suffit de mettre la position de la ligne désirée avant la virgule.

```
# Extraction d'une ligne
A[2, ]
```

```
## [1]  2  8 14 20
```


Pour accéder à une colonne complète, il suffit de mettre la position de la colonne désirée après la virgule.

```
# Extraction d'une colonne
A[, 3]
```

```
## [1] 13 14 15 16 17 18
```

Pour accéder à un groupe d'éléments, il faut déterminer l'intervalle des lignes et des colonnes désirées.

```
# Extraction de quelques éléments regroupées
A[3:5, 2:3]
```

```
##      [,1] [,2]
## [1,]    9   15
## [2,]   10   16
## [3,]   11   17
```

3.1.5 Dataframe

Un jeu de données se structure sous forme d'un tableau dans lequel chaque ligne correspond à une observation (individu) et chaque colonne à une caractéristique (variable). Les data frame sont les objets les plus utilisées lors de l'analyse d'une base de données. Contrairement aux vecteurs et aux matrices, une dataframe peut avoir différents type de variables (numérique, logique et chaînes de caractères). La fonction `data.frame()` permet la création de la base de données.

```
dataframe <- data.frame(
  ID = 1:5,
  Genre = c("Homme", "Femme", "Femme", "Femme", "Homme"),
  Age = c(45, 42, 45, 43, 44)
)
dataframe
```

```
##   ID Genre Age
## 1  1 Homme  45
## 2  2 Femme  42
## 3  3 Femme  45
## 4  4 Femme  43
## 5  5 Homme  44
```

Les colonnes d'une dataframe sont toujours nommées et correspondent à la variable mesurée. Les lignes sont automatiquement numérotées par ordre.

La fonction `str()` permet d'afficher la structure de la dataframe en affichant le nom de la variable, le type de celle-ci ainsi que les valeurs des observations.

```
# Structure
str(dataframe)
```

```
## 'data.frame':    5 obs. of  3 variables:
## $ ID      : int  1 2 3 4 5
## $ Genre: chr  "Homme" "Femme" "Femme" "Femme" ...
## $ Age     : num  45 42 45 43 44
```

La fonction `View()` permet de visionner la data frame dans une autre fenêtre.

```
# Structure
View(dataframe)
```

Afin d'analyser les données, il est important de pouvoir d'en extraire uniquement une partie. Il existe deux façons d'extraire une colonne. La première consiste à reproduire le cas de la matrice en sélectionnant la position de la colonne.

```
# Extraction de colonnes
dataframe[, 2]
```

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

La deuxième option est d'utiliser le symbole `$`. Il doit être placé entre le nom de la data frame et le nom de la colonne. **Il est conseillé d'utiliser cette option pour extraire une colonne d'une data frame.**

```
# Extraction de colonnes
dataframe$Genre
```

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

Pour extraire une ligne de la base de donnée, il faut procéder comme pour la matrice.

```
# Extraction de ligne
dataframe[2, ]
```

```
##   ID Genre Age
## 2   2 Femme 42
```

Pour extraire les observations (lignes) qui possèdent certaines caractéristiques, il est possible d'écrire la ligne suivante comme suit:

```
# Extraction de ligne
dataframe[dataframe$Genre == "Homme", ]
```

```
##   ID Genre Age
## 1   1 Homme 45
## 5   5 Homme 44
```

Il est également possible de mettre plusieurs conditions.

```
# Extraction de ligne
dataframe[dataframe$Genre == "Femme" & dataframe$Age < 44, ]
```

```
##   ID Genre Age
```

```
## 2 2 Femme 42
## 4 4 Femme 43
```

Il est possible d'ajouter une colonne à la base de données. Plusieurs options sont possibles:

1. Créer un vecteur de même taille que la longueur de la base de données et l'ajouter à la base de données en utilisant la fonction `cbind()`.
2. Créer une nouvelle variable directement dans la base de données en déterminant son nom grâce au signe `$`.

```
yeux <- c("brun", "brun", "bleu", "bleu", "brun")
dataframe <- cbind(dataframe, yeux)

dataframe$cheveux <- c("blond", "brun", "blond", "noir", "noir")

str(dataframe)
```

```
## 'data.frame': 5 obs. of 5 variables:
## $ ID : int 1 2 3 4 5
## $ Genre : chr "Homme" "Femme" "Femme" "Femme" ...
## $ Age : num 45 42 45 43 44
## $ yeux : chr "brun" "brun" "bleu" "bleu" ...
## $ cheveux: chr "blond" "brun" "blond" "noir" ...
```

3.2 Opérateurs logiques

Opérateur	Description
<	strictement inférieur
<=	inférieur ou égal
>	strictement supérieur
>=	supérieur ou égal
==	égal
!=	différent
!x	non x
x y	x ou y
x & y	x et y

Chapter 4

Packages et données

Ce premier chapitre introduit deux concepts importants dans R. Le premier est les packages et le second concerne les données.

4.1 Installation et gestion des packages

Les packages sont des regroupements de fonctions et de jeux de données développés dans R et qui doivent se télécharger une seule fois, mais ils devront être importés à chaque utilisation. Le code ci-dessous permet d'installer le package `ggplot2`:

```
install.packages("ggplot2", dependencies = TRUE)
```

Avant chaque utilisation des packages, il est nécessaire d'importer le package grâce au code suivant:

```
library(ggplot2)
```

4.2 Téléchargement des données

Dans R, les bases de données se déclinent de plusieurs façons:

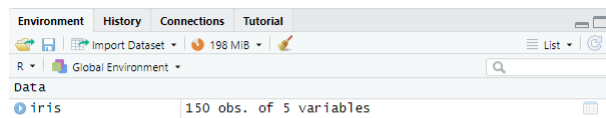
1. Les bases de données peuvent être directement incluses dans R ou dans les packages.
2. Les bases de données peuvent être créées dans l'environnement sauvees dans l'environnement R. Ces fichiers ont une extension `.RData`
3. Les bases de données peuvent être issues de fichiers externes. Ces fichiers peuvent avoir différentes extensions, les plus courantes étant `.csv` et `.txt`.

4.2.1 Bases de données issues de la base de R ou des packages

Le code suivant permet d'importer le jeu de données “iris” disponible de base dans R.

```
data(iris)
```

Un objet `iris` apparaît dans l'environnement du projet comme le montre la figure suivante.



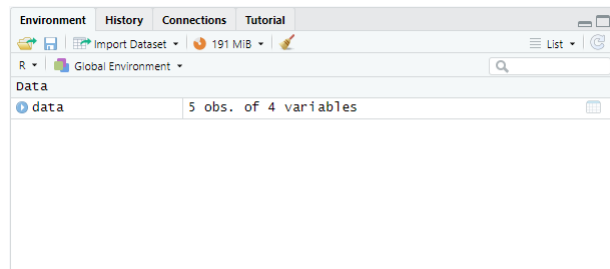
Si les données sont dans un package, le package doit être importé au préalable.

4.2.2 Bases de données issues dans un fichier .RData

Les fichiers `.RData` sont spécifiques au langage R et peuvent contenir plusieurs objets en un seul fichier. Pour importer des données, il faut utiliser la fonction `load()`

```
load("04-data.RData")
```

Tous les objets importés sont chargés dans l'environnement de travail.



4.2.3 Bases de données issues de fichiers externes

Les données à analyser sont souvent disponibles dans un fichier externe sous différents formats tels que `.csv` ou `.txt`. Pour importer ces données, il existe une fonction par type de fichier (`read.csv()` et `read.table()`). Lorsqu'une de ces fonctions est utilisée, le contenu est stocké dans une dataframe. Il est nécessaire de spécifier le chemin d'accès entre votre logiciel et votre fichier à télécharger.

Vous pouvez le faire de deux manières: 1. En utilisant la fonction `setwd()` et en spécifiant à l'intérieur la direction complète qui va permettre au logiciel de retrouver votre document dans vos fichiers. 1. En créant un projet dans lequel vous stockez vos documents. Ces fonctions acceptent des arguments qui permettent de s'adapter à la nature de fichier à importer. Parmi ces arguments, il y en a trois principaux:

1. **header** qui est une valeur logique (**TRUE** ou **FALSE**) pour la présence d'un en-tête avec les noms de variables. Cet argument est mis par défaut à **TRUE** pour la fonction `read.csv()` et à **FALSE** pour la fonction `read.table()`.
2. **sep** qui est le caractère dont les champs sont séparés. Cet argument est mis par défaut à `,`.
3. **dec** qui est le séparateur décimal. Cet argument est mis par défaut à `.`.

Le seul argument obligatoire est le chemin d'accès au fichier à lire. Il n'est pas nécessaire de spécifier le chemin complet si le fichier à lire se trouve dans le dossier du projet. Si ce n'est pas le cas, vous devez spécifier le chemin d'accès complet à partir de ce qui a été fait avec la fonction `setwd()`. Pour charger une base de donnée nommé `04-data.csv` dont les valeurs sont séparées par des virgules, il suffit d'écrire la ligne suivante:

```
csv_data <- read.csv("04-data.csv")
csv_data
```

```
##      Nom Age Note.1 Note.2
## 1  Marc  18   5.0   5.50
## 2  Anne  20   6.0   4.00
## 3 Marie  21   4.5   4.75
## 4  Jean  17   3.5   5.00
## 5 Sophie 18   5.0   4.75
```

Pour charger une base de donnée nommé `04-data.txt` dont les valeurs sont séparées par des points-virgules, il suffit d'écrire comme dans la ligne suivante en n'oubliant pas de spécifier le caractère de séparation avec l'argument `sep=";"`.

```
txt_data <- read.table("04-data.txt", sep = ";", header = TRUE)
txt_data
```

```
##      Nom Age Note.1 Note.2
## 1  Marc  18   5.0   5.50
## 2  Anne  20   6.0   4.00
## 3 Marie  21   4.5   4.75
## 4  Jean  17   3.5   5.00
## 5 Sophie 18   5.0   4.75
```

Il est également possible de charger une base de données issues d'un fichier externe en l'important depuis le menu `File > Import Dataset > From Text` puis de sélectionner le fichier dans vos dossiers.

Chapter 5

Description des données quantitatives et qualitatives

Ce deuxième chapitre regroupe quelques fonctions pour décrire numériquement et graphiquement les variables quantitatives et qualitatives. Des fonctions spécifiques doivent être utilisées relativement à la nature de la variable.

```
df <- read.csv("sport.csv", sep = ";", header = TRUE)
str(df)
```

```
## 'data.frame': 15 obs. of 7 variables:
## $ Identifiant : int 314 323 547 336 678 442 667 890 426 789 ...
## $ Age : int 21 17 18 22 29 72 34 28 75 48 ...
## $ Poids : num 50.5 46 57.7 52 60 87 66 58.2 89 77 ...
## $ Grandeur : int 166 159 170 177 178 182 176 163 179 165 ...
## $ Satisfaction: chr "extremement_satisfait" "moyennement_satisfait" "plutot_insatisfait" "peu_satisfait"
## $ Sport : int 180 120 60 30 200 0 180 60 90 320 ...
## $ Sexe : chr "F" "F" "F" "M" ...
```

On observe qu’il existe plusieurs types de variables.

5.1 Description numérique

5.1.1 Variables qualitatives

La variable “Sexe” est une variable qualitative nominale. Lorsque l’on regarde dans la base de données, on observe qu’elle est considérée comme une chaîne de caractères. Il est dès lors possible de la transformer en facteur. Une situation similaire arrive avec la variable “Satisfaction” qui est une variable qualitative ordinale. Il est également possible de la transformer afin d’obtenir un facteur ordonné. Pour les deux cas, la transformation s’effectue grâce à la fonction

`factor()`. Lorsque la variable est ordinale, des arguments doivent être ajoutés afin de préciser qu'il s'agit d'un facteur ordonné et de donner l'ordre des niveaux.

```
df$Sexe <- factor(df$Sexe)
df$Sexe

## [1] F F F M M M M F M F F F M F
## Levels: F M

df$Satisfaction <- factor(df$Satisfaction, ordered = TRUE,
                          levels = c("pas_du_tout_satisfait", "plutot_insatisfait",
                                      "moyennement_satisfait", "tres_satisfait",
                                      "extremement_satisfait"))
df$Satisfaction

## [1] extremement_satisfait moyennement_satisfait plutot_insatisfait
## [4] pas_du_tout_satisfait tres_satisfait pas_du_tout_satisfait
## [7] moyennement_satisfait plutot_insatisfait moyennement_satisfait
## [10] tres_satisfait plutot_insatisfait pas_du_tout_satisfait
## [13] plutot_insatisfait tres_satisfait tres_satisfait
## 5 Levels: pas_du_tout_satisfait < ... < extremement_satisfait
```

Pour les variables qualitatives, la description numérique classique est le tableau de contingence que l'on obtient avec la fonction `table()`. Il est possible de décrire chaque variable à la fois ou de décrire une variable en fonction d'une autre.

```
table(df$Sexe)

##
## F M
## 9 6

table(df$Satisfaction)

##
## pas_du_tout_satisfait plutot_insatisfait moyennement_satisfait
## 3 4 3
## tres_satisfait extremement_satisfait
## 4 1

table(df$Sexe, df$Satisfaction)

##
## pas_du_tout_satisfait plutot_insatisfait moyennement_satisfait
## F 1 4 1
## M 2 0 2
##
## tres_satisfait extremement_satisfait
## F 2 1
```

```
##      M                2                0
```

5.1.2 Variables quantitatives

Les variables quantitatives sont généralement décrites à l'aide de:

1. Mesures de tendance centrale: mode, moyenne, médiane
2. Mesures de dispersion: étendue, espace interquartile, variance et écart-type

La moyenne et la médiane peuvent se calculer directement avec les fonctions `mean()` et `median()` respectivement. Le mode d'une variable peut s'obtenir en analysant le tableau de contingence. Dans notre cas, on observe qu'il n'y a pas de mode étant donné qu'aucun âge apparaît plusieurs fois.

```
mean(df$Age)
```

```
## [1] 34.66667
```

```
median(df$Age)
```

```
## [1] 29
```

```
table(df$Age)
```

```
##
```

```
## 17 18 20 21 22 25 28 29 32 34 35 44 48 72 75
```

```
##  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
```

L'étendue d'une variable peut s'obtenir en soustrayant la valeur maximale et la valeur minimale d'une variable. La variance et l'écart-type s'obtiennent directement avec les fonctions `var()` et `sd()` respectivement. L'espace interquartile, utilisé pour construire le boxplot, s'obtient grâce à la fonction `IQR()`.

```
min(df$Age)
```

```
## [1] 17
```

```
max(df$Age)
```

```
## [1] 75
```

```
max(df$Age)-min(df$Age)
```

```
## [1] 58
```

```
var(df$Age)
```

```
## [1] 329.6667
```

```
sd(df$Age)
```

```
## [1] 18.15673
```

```
IQR(df$Age)
```

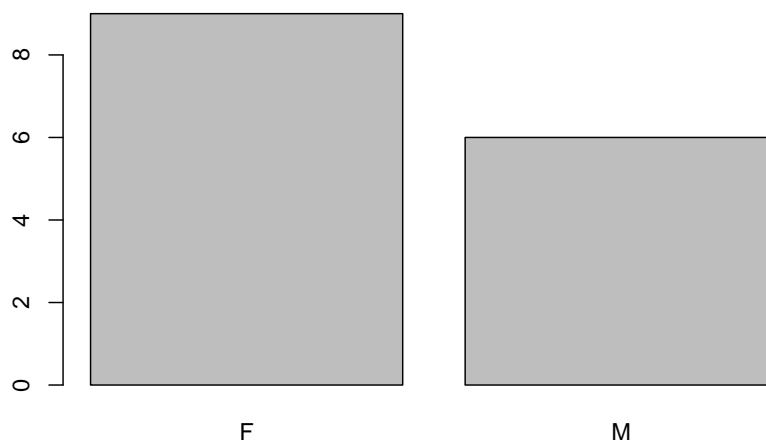
```
## [1] 18
```

5.2 Description graphique

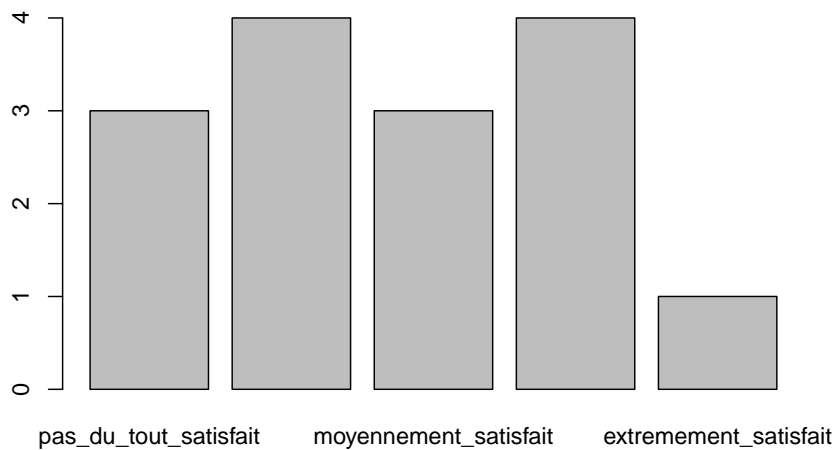
5.2.1 Variables qualitatives

Les variables qualitatives peuvent être représentées grâce à des graphiques en barre ou des graphiques en camembert. Les fonctions `barplot()` et `pie()` permettent de représenter les tableaux de contingence. Pour rappel, il est possible de sauver un objet dans l’environnement et de l’utiliser ensuite directement pour faire les graphes. C’est ce qui est proposé pour le tableau de contingence de la variable “Satisfaction”.

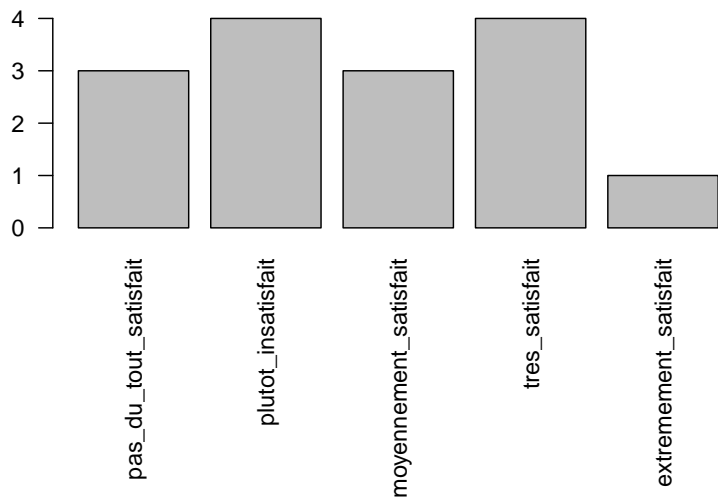
```
barplot(table(df$Sexe))
```



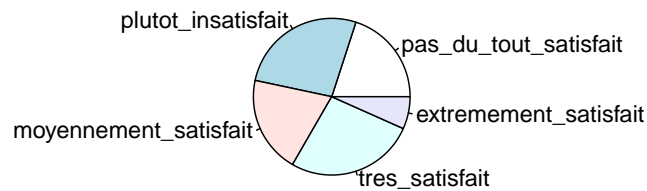
```
tableSatisfaction <- table(df$Satisfaction)
barplot(tableSatisfaction)
```



```
par(mar=c(11,4,4,4))
barplot(tableSatisfaction, las=2)
```



```
pie(tableSatisfaction)
```



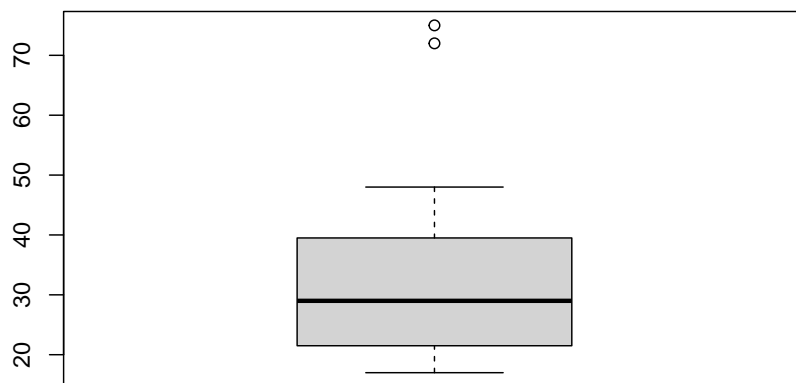
5.2.2 Variables quantitatives

Les variables quantitatives peuvent être représentées grâce à des histogrammes ou des boxplots. Les histogrammes permettent de représenter la fréquence de chaque valeur de la variable, tandis que les boxplots représente la distribution de la variable à l'aide des quartiles. Il est possible de spécifier des arguments dans la fonction graphiques afin de donner un titre ou de nommer un axe.

```
hist(df$Age, main="Histogramme de l'age", xlab = "Age")
```



```
boxplot(df$Age)
```



Chapter 6

Distribution de probabilités

Ce chapitre couvre les fonctions relatives aux distributions de probabilité. Pour rappel, chaque variable aléatoire a une distribution de probabilité.

Dans ce cours, plusieurs distributions sont abordées, mais la principale est la distribution normale.

6.1 Distribution normale

La distribution normale est caractérisée par deux paramètres, la moyenne et l'écart-type. La variation de ces deux paramètres implique que plusieurs distributions normales existent. Lorsque la moyenne vaut 0 et l'écart-type 1, on parle de distribution normale centrée-réduite. Il s'agit de la distribution normale avec laquelle il est commun de travailler.

Plusieurs fonctions existent pour travailler avec la distribution normale: `dnorm()`, `pnorm()`, `qnorm()`, et `rnorm()`.

La fonction `dnorm()` permet de calculer la densité pour n'importe quelle valeur de `x`. Par exemple, calculons la densité pour une valeur de 3 avec une distribution normale avec les paramètres de moyenne de 4, et d'écart-type de 2. Calculons ensuite la même densité mais pour une distribution normale centrée-réduite. Pour ce cas, on remarque que les résultats sont similaires lorsque les paramètres sont spécifiés et lorsqu'ils ne le sont pas. Ceci s'explique par le fait que les paramètres de la distribution normale centrée-réduite sont les paramètres par défaut de la fonction.

```
x <- 3
dnorm(x, mean = 4, sd = 2)
```

```
## [1] 0.1760327
```

```
dnorm(x, mean = 0, sd = 1)
```

```
## [1] 0.004431848
```

```
dnorm(x)
```

```
## [1] 0.004431848
```

Il est également possible de calculer la densité entre deux valeurs. Calculons la densité entre -4 et 4 pour une distribution normale avec une moyenne de 3 et un écart-type de 2.

```
x <- seq(-4, 4, by=1)
x
```

```
## [1] -4 -3 -2 -1 0 1 2 3 4
```

```
dnorm(x, mean = 3, sd = 2)
```

```
## [1] 0.0004363413 0.0022159242 0.0087641502 0.0269954833 0.0647587978
```

```
## [6] 0.1209853623 0.1760326634 0.1994711402 0.1760326634
```

```
sum(dnorm(x, mean = 3, sd = 2))
```

```
## [1] 0.7756925
```

La fonction `pnorm` donne la fonction de distribution de la loi normale. La fonction de distribution cumulative (CDF) correspond à la probabilité que la variable X prenne une valeur inférieure ou égale à x . La fonction de survie correspond à la probabilité que la variable X prenne une valeur supérieure à x . La fonction `pnorm` permet de calculer ces deux fonctions en manipulant le paramètre `lower.tail`. Lorsque `lower.tail = TRUE`, ce qui est le paramètre par défaut, la CDF est obtenue. Lorsque `lower.tail = FALSE`, la fonction de survie est obtenue. Prenons le quantile 1 pour une loi normale centrée-réduite et calculons ces deux fonctions.

```
pnorm(1, mean = 0, sd = 1)
```

```
## [1] 0.8413447
```

```
pnorm(1, mean = 0, sd = 1, lower.tail = FALSE)
```

```
## [1] 0.1586553
```

```
pnorm(1, mean = 0, sd = 1) + pnorm(1, mean = 0, sd = 1, lower.tail = FALSE)
```

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

Si nous sommes la probabilité d'être plus petit qu'un quantile donné et la probabilité d'être plus grand que ce même quantile, le résultat est de 1 parce que tout l'espace possible de la distribution est couvert.

```
pnorm(1, mean = 0, sd = 1) + pnorm(1, mean = 0, sd = 1, lower.tail = FALSE)
```

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

Comme la distribution normale centrée-réduite est symétrique en 0, la CDF est partagée en deux autour de cette valeur.

```
pnorm(0, mean = 0, sd = 1)
```

```
## [1] 0.5
```

```
pnorm(0, mean = 0, sd = 1, lower.tail = FALSE)
```

```
## [1] 0.5
```

La fonction `qnorm` permet d'obtenir le quantile pour n'importe quelle probabilité. Pour rappel, un quantile est une valeur qui divise les données d'une distribution en segments de même fréquence. Dans le cadre d'une fonction de distribution, il indique la valeur en dessous de laquelle une certaine proportion des données se situe. Cette fonction calcule l'inverse de la fonction `pnorm`. Pour simplifier la compréhension, reprenons le dernier exemple présenté.

```
pnorm(0, mean = 0, sd = 1)
```

```
## [1] 0.5
```

```
qnorm(0.5, mean = 0, sd = 1)
```

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

La fonction `qnorm` permet de retrouver la valeur correspondant à une probabilité, alors que la fonction `pnorm` permet de calculer la probabilité à partir d'une valeur.

La fonction `rnorm` permet de générer n observations à partir d'une distribution normale. Il s'agit d'une fonction particulièrement utilisée pour la simulation. Comme les autres fonctions concernant la distribution normale, les paramètres de base simuler des données avec une moyenne de 0 et un écart-type de 1. Il est possible de modifier ces paramètres comme suit:

```
set.seed(1234) #permet de garantir la reproductibilité des analyses
rnorm(100)
```

```
## [1] -1.207065749 0.277429242 1.084441177 -2.345697703 0.429124689
## [6] 0.506055892 -0.574739960 -0.546631856 -0.564451999 -0.890037829
## [11] -0.477192700 -0.998386445 -0.776253895 0.064458817 0.959494059
## [16] -0.110285494 -0.511009506 -0.911195417 -0.837171680 2.415835178
## [21] 0.134088220 -0.490685897 -0.440547872 0.459589441 -0.693720247
## [26] -1.448204910 0.574755721 -1.023655723 -0.015138300 -0.935948601
## [31] 1.102297546 -0.475593079 -0.709440038 -0.501258061 -1.629093469
## [36] -1.167619262 -2.180039649 -1.340993192 -0.294293859 -0.465897540
```

```
## [41] 1.449496265 -1.068642724 -0.855364634 -0.280623002 -0.994340076
## [46] -0.968514318 -1.107318193 -1.251985886 -0.523828119 -0.496849957
## [51] -1.806031257 -0.582075925 -1.108889624 -1.014962009 -0.162309524
## [56] 0.563055819 1.647817473 -0.773353424 1.605909629 -1.157808548
## [61] 0.656588464 2.548991071 -0.034760390 -0.669633580 -0.007604756
## [66] 1.777084448 -1.138607737 1.367827179 1.329564791 0.336472797
## [71] 0.006892838 -0.455468738 -0.366523933 0.648286568 2.070270861
## [76] -0.153398412 -1.390700947 -0.723581777 0.258261762 -0.317059115
## [81] -0.177789958 -0.169994077 -1.372301886 -0.173787170 0.850232257
## [86] 0.697608712 0.549997351 -0.402731975 -0.191593770 -1.194527880
## [91] -0.053158819 0.255196001 1.705964007 1.001513252 -0.495583443
## [96] 0.355550297 -1.134608044 0.878203627 0.972916753 2.121117105
```

Comme les autres fonctions concernant la distribution normale, les paramètres de base simuler des données avec une moyenne de 0 et un écart-type de 1. Il est possible de modifier ces paramètres comme suit par exemple:

```
set.seed(1234)
rnorm(100, mean = 3, sd = 1.5)
```

```
## [1] 1.1894014 3.4161439 4.6266618 -0.5185466 3.6436870 3.7590838
## [7] 2.1378901 2.1800522 2.1533220 1.6649433 2.2842110 1.5024203
## [13] 1.8356192 3.0966882 4.4392411 2.8345718 2.2334857 1.6332069
## [19] 1.7442425 6.6237528 3.2011323 2.2639712 2.3391782 3.6893842
## [25] 1.9594196 0.8276926 3.8621336 1.4645164 2.9772925 1.5960771
## [31] 4.6534463 2.2866104 1.9358399 2.2481129 0.5563598 1.2485711
## [37] -0.2700595 0.9885102 2.5585592 2.3011537 5.1742444 1.3970359
## [43] 1.7169530 2.5790655 1.5084899 1.5472285 1.3390227 1.1220212
## [49] 2.2142578 2.2547251 0.2909531 2.1268861 1.3366656 1.4775570
## [55] 2.7565357 3.8445837 5.4717262 1.8399699 5.4088644 1.2632872
## [61] 3.9848827 6.8234866 2.9478594 1.9955496 2.9885929 5.6656267
## [67] 1.2920884 5.0517408 4.9943472 3.5047092 3.0103393 2.3167969
## [73] 2.4502141 3.9724299 6.1054063 2.7699024 0.9139486 1.9146273
## [79] 3.3873926 2.5244113 2.7333151 2.7450089 0.9415472 2.7393192
## [85] 4.2753484 4.0464131 3.8249960 2.3959020 2.7126093 1.2082082
## [91] 2.9202618 3.3827940 5.5589460 4.5022699 2.2566248 3.5333254
## [97] 1.2980879 4.3173054 4.4593751 6.1816757
```

6.2 Autres distributions

...à venir

Chapter 7

Les boucles FOR

Les boucles sont utilisées en programmation pour exécuter automatiquement un bloc de code plusieurs fois, ce qui permet d'éviter la répétition manuelle. Elles sont très utiles pour effectuer des calculs répétitifs ou parcourir des vecteurs, des listes, des matrices ou des bases de données.

Dans ce chapitre, nous traitons des boucles `for`. D'autres types de boucles existent. Les boucles `for` permettent d'itérer sur une séquence de valeurs prédéfinies en exécutant le code présent à l'intérieur de la boucle pour chaque élément.

7.1 Structure

La structure générale d'une boucle `for` comprend:

1. La fonction `for`
2. La variable de répétition
3. Le vecteur d'intérêt
4. Les instructions à exécuter

```
# for(variable in vecteur) {  
#   # Instructions à exécuter  
# }
```

Cette structure est systématiquement la même, mais sa complexité peut varier en fonction des tâches que l'on souhaite effectuer.

7.2 Utilisation

Les boucles `for` ont plusieurs utilités en fonction de ce que l'on souhaite effectuer. Nous présentons ici les principales façons d'utiliser les boucles et les fonctions régulièrement utilisées à l'intérieur des boucles.

7.2.1 Afficher

La fonction la plus simple d'une boucle `for` est l'affichage. L'exemple proposé ci-dessous permet de comprendre comment la boucle fonctionne. Nous souhaitons afficher les nombres de 1 à 10. Par conséquent, le **vecteur** qui nous intéresse doit contenir toutes ces valeurs. La **variable** que nous spécifions pour la boucle est l'indice `i`. Bien sur, cet indice pourrait se nommer différemment, mais il est d'usage de l'utiliser. La fonction dont nous avons besoin pour afficher les nombres est `print()`. Elle apparaît donc comme l'instruction de la fonction. On observe qu'il est possible d'indiquer le vecteur directement dans la boucle comme dans le premier exemple ou d'avoir créé une variable au préalable comme dans le second exemple.

```
for(i in 1:10) {  
  print(i)  
}
```

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

```
nbre <- c(1:10)  
print(nbre)
```

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

```
for(i in nbre) {  
  print(i)  
}
```

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

7.2.2 Itérer

Les boucles `for` permettent également d'itérer la même instruction plusieurs fois sans devoir répéter le code. Dans l'exemple, nous souhaitons afficher à plusieurs reprises la phrase "Aujourd'hui, nous sommes" avec le jour en question à la fin de la phrase. Si nous effectuions le code séparément, nous aurions besoin de copier le code 7 fois. En analysant ces répétitions, nous pouvons identifier une partie du texte qui ne change pas, à savoir "Aujourd'hui, nous sommes" tandis que l'autre partie, à savoir les jours de la semaine, change à chaque répétitions. Nous souhaitons donc concaténer la partie qui change à la phrase qui reste similaire au fur et à mesure des répétitions. Par conséquent, nous fixons comme **variable** l'index `iet` comme **vecteur** les jours. La fonction `print` permet d'afficher le résultat. Cependant, une nouvelle fonction est nécessaire, à savoir `paste` pour concaténer les deux parties.

```
print("Aujourd'hui, nous sommes lundi")

## [1] "Aujourd'hui, nous sommes lundi"
print("Aujourd'hui, nous sommes mardi")

## [1] "Aujourd'hui, nous sommes mardi"
print("Aujourd'hui, nous sommes mercredi")

## [1] "Aujourd'hui, nous sommes mercredi"
print("Aujourd'hui, nous sommes jeudi")

## [1] "Aujourd'hui, nous sommes jeudi"
print("Aujourd'hui, nous sommes vendredi")

## [1] "Aujourd'hui, nous sommes vendredi"
print("Aujourd'hui, nous sommes samedi")

## [1] "Aujourd'hui, nous sommes samedi"
print("Aujourd'hui, nous sommes dimanche")

## [1] "Aujourd'hui, nous sommes dimanche"
```

Nous allons donc créer un vecteur contenant tous les jours de la semaine. Ensuite, nous devons créer la boucle.

```
jours <- c("lundi", "mardi", "mercredi", "jeudi", "vendredi")

for(i in jours) {
  #concaténer et afficher
}
```

```
paste("Aujourd'hui, nous sommes", "lundi") #comprendre le fonctionnement de paste
```

```
## [1] "Aujourd'hui, nous sommes lundi"
for(i in jours) {
  print(paste("Aujourd'hui, nous sommes", jours))
}
```

```
## [1] "Aujourd'hui, nous sommes lundi"      "Aujourd'hui, nous sommes mardi"
## [3] "Aujourd'hui, nous sommes mercredi"   "Aujourd'hui, nous sommes jeudi"
## [5] "Aujourd'hui, nous sommes vendredi"
## [1] "Aujourd'hui, nous sommes lundi"      "Aujourd'hui, nous sommes mardi"
## [3] "Aujourd'hui, nous sommes mercredi"   "Aujourd'hui, nous sommes jeudi"
## [5] "Aujourd'hui, nous sommes vendredi"
## [1] "Aujourd'hui, nous sommes lundi"      "Aujourd'hui, nous sommes mardi"
## [3] "Aujourd'hui, nous sommes mercredi"   "Aujourd'hui, nous sommes jeudi"
## [5] "Aujourd'hui, nous sommes vendredi"
## [1] "Aujourd'hui, nous sommes lundi"      "Aujourd'hui, nous sommes mardi"
## [3] "Aujourd'hui, nous sommes mercredi"   "Aujourd'hui, nous sommes jeudi"
## [5] "Aujourd'hui, nous sommes vendredi"
```

7.2.3 Créer un vecteur

Les boucles `for` permettent également de créer de manière automatisée des vecteurs, des matrices, des listes ou des bases de données. Dans notre exemple, nous voulons créer un vecteur contenant le carré des nombres. Il existe plusieurs façons de remplir le vecteur, soit avec la fonction `c()` comme dans le premier exemple, soit avec la fonction `append()` comme dans le deuxième exemple. De plus, l'opération souhaitée peut se faire directement au moment de remplir le vecteur comme dans le premier exemple ou dans une ligne de code séparée comme dans le deuxième exemple. Nous vous conseillons de procéder de manière séquentielle lorsque les instructions se complexifient.

```
nbre <- c(1:10)
carre <- vector() #vecteur vide

for(i in nbre) {
  carre <- c(carre, i^2)
}

print(carre)
```

```
## [1] 1 4 9 16 25 36 49 64 81 100
```



```
carre <- vector()

for(i in nbre) {
  squared <- (i^2)
  carre <- append(carre, squared)
}
```

Il est également utile de savoir comment indexer les variables afin de pouvoir modifier des vecteurs. Dans le premier exemple, nous créer un nouveau vecteur qui correspond à un précédent vecteur plus 1. Le développement proposé avant la création de la boucle permet de comprendre comment procéder pour la mise en place des instructions dans la boucle. La manière proposée pour ajouter les valeur dans le vecteur utilise uniquement les index. Nous pouvons obtenir le même résultat en utilisant la fonction précédemment présentée `append()`

```
nbre <- c(1:10)
nbre[1]
```

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

```
nbre[1] + 1
```

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

```
nbrePLUS <- vector()
for (i in nbre) {
  nbrePLUS[i] <- nbre[i] + 1 # augmente chaque nombre de 1
}
print(nbrePLUS)
```

```
## [1] 2 3 4 5 6 7 8 9 10 11
```

```
nbrePLUS <- vector()
for (i in nbre) {
  addONE <- nbre[i] + 1 # augmente chaque nombre de 1
  nbrePLUS <- append(nbrePLUS, addONE)
}
print(nbrePLUS)
```

```
## [1] 2 3 4 5 6 7 8 9 10 11
```

Jusqu'à présent, nous avons itéré sur un vecteur directement en parcourant les éléments du vecteur. Il est également possible d'itérer sur un vecteur en accédant aux éléments de ce dernier à travers les indices en utilisant `1:length(vecteur)`. Cette méthode nécessite de bien faire attention au fait que le vecteur ne soit pas vide. Dans le deuxième exemple, nous proposons d'itérer selon cette deuxième approche en ajoutant 0.5 aux notes proposées.

```
notes <- c(5.5, 4.5, 5, 3.5, 2.5)

for (i in 1:length(notes)) {
  notes[i] <- notes[i] + 0.5 # augmente chaque note de 0.5 point
}

print(notes)

## [1] 6.0 5.0 5.5 4.0 3.0
```

7.2.4 Parcourir une matrice

A venir...