Guide R

Prof. Audrey Bürki, Samuel Arthers, Mégane Bollenrücher

2025-05-23

Contents

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 le cours de Stat I.

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 controler 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

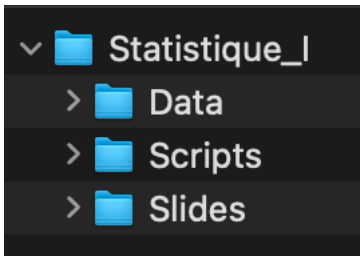
   ii. Installer le logiciel R sur votre ordinateur en exécutant le fichier téléchargé.
2. Installer RStudio sur le site suivant: https://posit.co/download/rstudio-desktop/

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 Organiser ses fichiers

Afin de gérer vos scripts et vos bases de données, nous vous proposons d'organiser vos fichiers au sein de dossiers étiquetés.

Vous pourriez par exemple créer un dossier "Statistique_I" et soit y placer tous vos fichiers (jeux de données, scripts) dont vous avez besoin pour le cours, ou faire des sous-dossier pour chaque thème. Vous pourriez, par exemple, suivre la structure suivante :



Attention, si vous manquez d'assiduité dans votre organisation, vous allez multiplier les possibilités de mauvaises manipulations.

2.4 Répertoire de travail

Lorsque vous travaillez dans R studio, le logiciel a besoin de savoir quel est votre répertoire de travail (ou working directory en anglais). Si vous ne le spécifiez pas vous-même, R studio va simplement en choisir un par défaut.

Afin de connaître le répertoire de travail actuel, vous pouvez utiliser la fonction *getwd( )*

```
getwd()
```

Nous vous conseillons de définir vous-même le répertoire de travail à chaque fois que vous commencez de travailler dans R, par exemple, en le spécifiant au début de vote script.

Si vous avez créé un dossier pour le cours, alors vous pouvez l'utiliser comme répertoire de travail. Cela a de nombreux avantages et notamment vous permet d'accéder facilement aux bases de données.

Pour définir ce répertoire de travail, deux alternatives sont à votre disposition, l'utilisation de la commande *setwd( )* ou passer par la création d'un projet R.

2.4.1 Définir son répertoire de travail avec *setwd( )*

La commande *setwd( )* vous permet de définir le répertoire de travail. Une bonne pratique consiste à l'indiquer au début du script si vous travaillez avec un script.

A chaque fois que vous retournez sur R studio, vous devrez spécifier votre répertoire de travail.

Pour ce faire, il suffit de copier le chemin qui mène au répertoire de travail à l'intérieur des parenthèses, comme dans l'exemple ci-dessous

```
setwd("~/Documents/assistanat/Statistique_I/")
```

RAPPEL, vous pouvez obtenir le chemin d'un dossier de la manière suivante:

- *Pour windows : Obtain file path in windows 10*
- *Pour MAC : Obtain file path on a Mac*
- *Pour linux : Obtain file path in Linux command line*

Une fois que vous avez indiqué votre répertoire de travail par la fonction *setwd( )*, si vous avez une base de données dans le répertoire de travail, vous pourrez l'accéder directement.

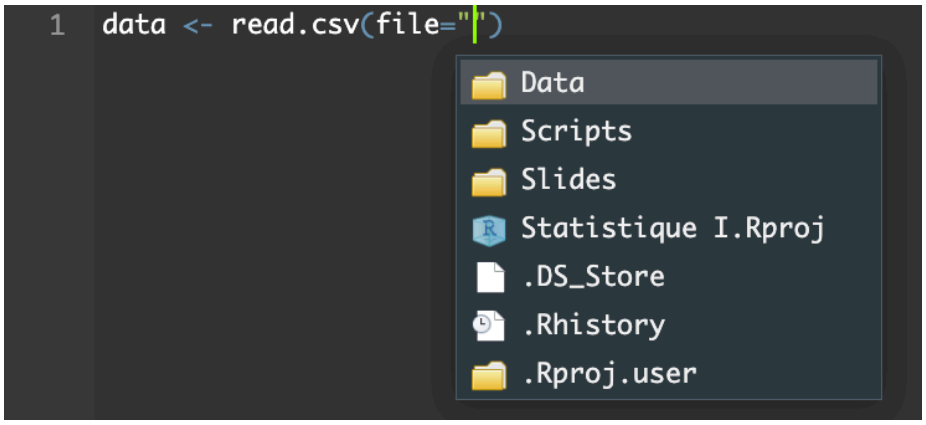
Exemple 1 :

```
# Par exemple, importer le fichier "data_ex1.csv",
# enregistré dans le répertoire de travail.
read.csv(file="data_ex1.csv")
```

Exemple 2 :

```
# Par exemple, importer le fichier "data_ex1.csv",
# enregistré dans le dossier Data qui se trouve dans le répertoire de travail.
read.csv(file="Data/data_ex1.csv")
```

Pour rappel, lorsque vous allez indiquer le chemin menant à un fichier, vous pouvez utiliser la touche 'tab' afin que Rstudio vous propose les différentes alternatives à votre disposition. Choisissez votre étape en cliquant dessus. La touche 'tab' peut être réutilisée jusqu'à ce que vous arriviez au fichier de votre choix. Cela vous permet d'éviter toute forme de fausse manipulation telle qu'une faute de frappe par exemple.

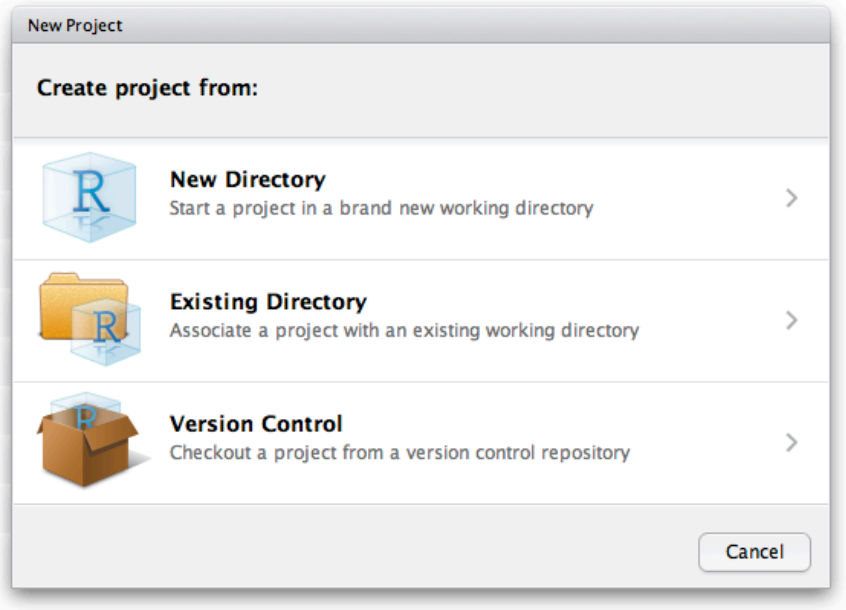


2.4.2 Projet R

A la place de préciser votre répertoire de travail par la fonction *setwd( )*, Rstudio vous propose une autre alternative: la creation d'un projet. La création d'un projet vous permet de générer un raccourci qui va lancer Rstudio directement depuis votre répertoire de travail choisi (p.ex, votre dossier Statistique_I). L'avantage d'ouvrir Rstudio depuis votre répertoire de travail est que cela change directement le point d'ancrage de vos chemins sans que vous ayez besoin de le spécifier par la fonction *setwd( )*.

Pour créer un projet, suivez les 5 étapes suivantes :

1. Générez un nouveau projet en suivant le chemin suivant dans votre interface RStudio: File > New project... (Fichier > Nouveau projet...)



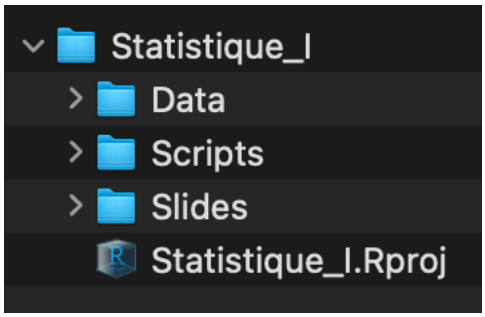
2. Vous avez ensuite deux possibilités:

   i. Créer un **nouveau** dossier
   Par exemple, créer un nouveau dossier appelé "Statistique_I" sur votre bureau ou ailleurs.

   ii. Générer le projet au sein d'un dossier **préexistant**
   Par exemple, instaurer le projet dans votre dossier "Statistique_I", rangé dans vos cours de première année.

3. L'étape suivante est de structurer votre répertoire de travail (nouveau ou existant) de la manière suivante:

   - **Statistique_I,** le nom de votre répertoire de travail choisi
     - **Statistique_I.Rproj**, votre fichier .Rproj que vous avez généré à l'étape précédente
     - **Data**, le dossier pour vos bases de données
     - **Scripts**, le dossier pour vos scripts
     - **Slides**, le dossier pour les slides du cours ou autres documents

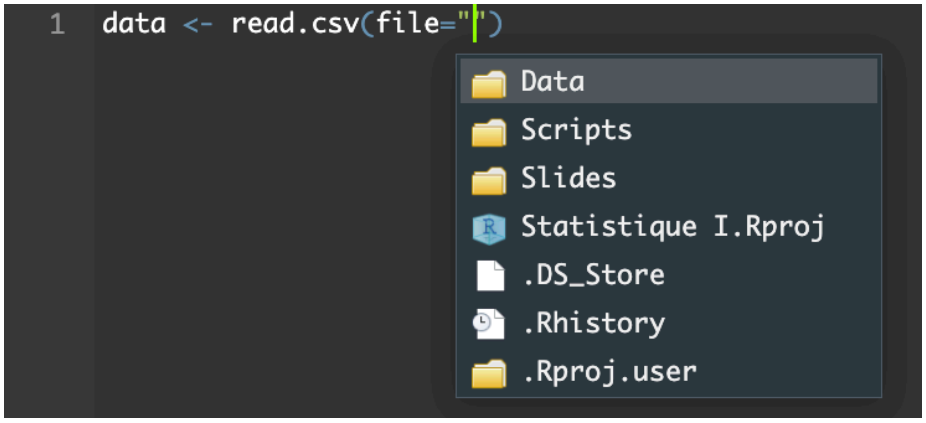


4. Vous pouvez maintenant ouvrir Rstudio en double-cliquant sur votre fichier Statistique_I.Rproj.

5. Une fois Rstudio ouvert par ce biais, les chemins menant à vos diverses bases de données seront directs.

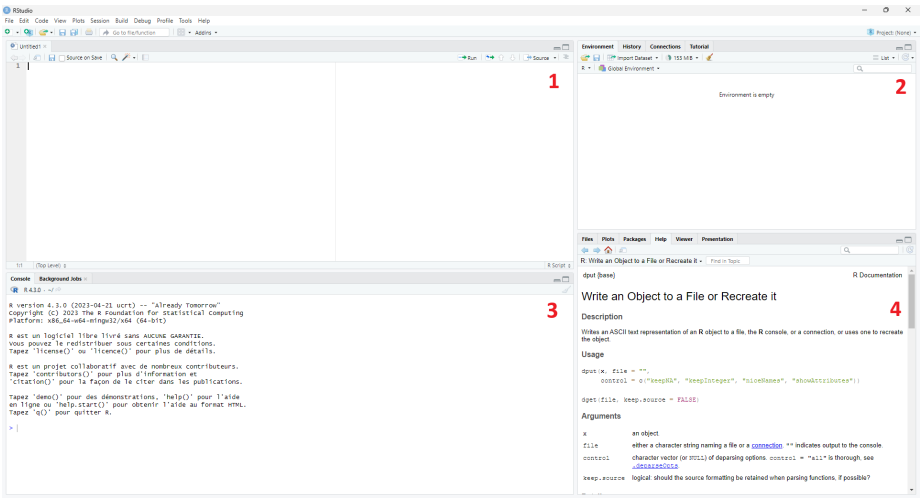
```
# Par exemple, importer le fichier "data_ex1.csv", enregistré dans le dossier Data
read.csv(file="Data/data_ex1.csv")
```

Pour rappel, lorsque vous allez indiquer le chemin menant à un fichier, vous pouvez utiliser la touche 'tab' afin que Rstudio vous propose les différentes alternatives à votre disposition. Choisissez votre étape en cliquant dessus. La touche 'tab' peut être réutilisée jusqu'à ce que vous arriviez au fichier de votre choix. Cela vous permet d'éviter toute forme de fausse manipulation telle qu'une faute de frappe par exemple.



2.5 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.

```
# Assignation 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 contenur 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 quantitée 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 fontion 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 chaine 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"): invalid factor level, NA
## generated
```

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'élément, 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 affichat 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 séléctionnant 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:

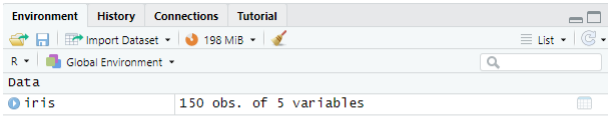
1. Les bases de données peuvent être directement incluses dans R ou dans les packages.
2. Les bases de données peuvent être créées dans l'environnement sauvées dans l'environnement R. Ces fichiers ont une extension .RData
3. Les bases de données peuvent être issues de fichiers externes. Ces fichiers 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` apparait dans l'environnement du projet comme le montre la figure suivante.



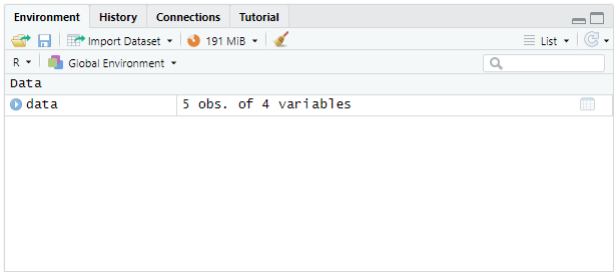
Si les données sont dans un package, le package doit être importé au préalable.

4.2.2 Bases de données issues dans un fichier .RData

Les fichiers `.RData` sont spécifique au language 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 sécifier 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  "extrememement_satisfait" "moyennement_satisfait" "plutot_insatisfait" "pa
##  $ 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 chaine 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 ajouté 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 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",
                                     "extrememement_satisfait"))
df$Satisfaction
```

```
##  [1] extrememement_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 < ... < extrememement_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 extrememement_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 extrememement_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)
```

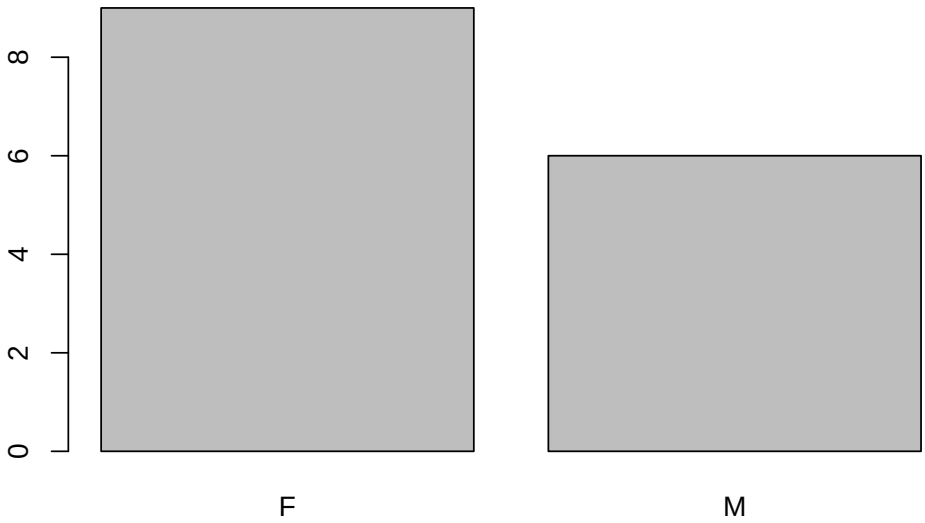
```
## [1] 18
```

5.2 Description graphique

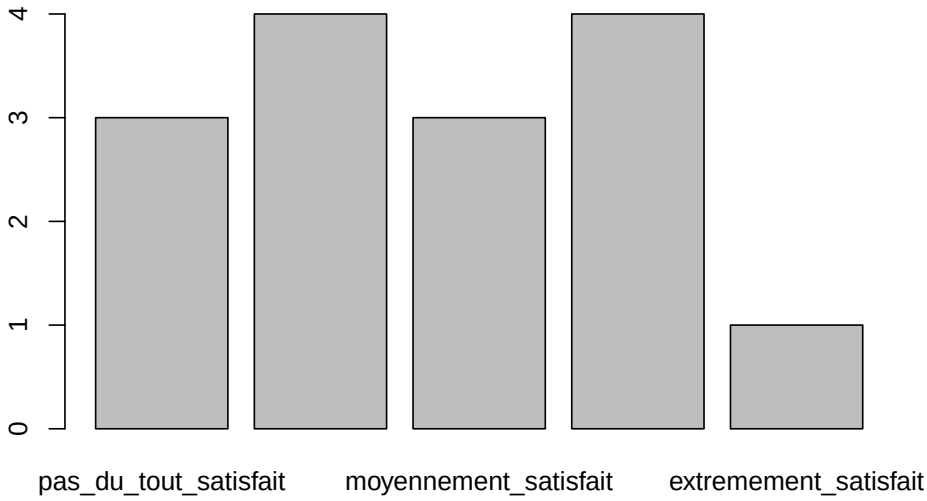
5.2.1 Variables qualitatives

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 contigence de la variable "Satisfaction".

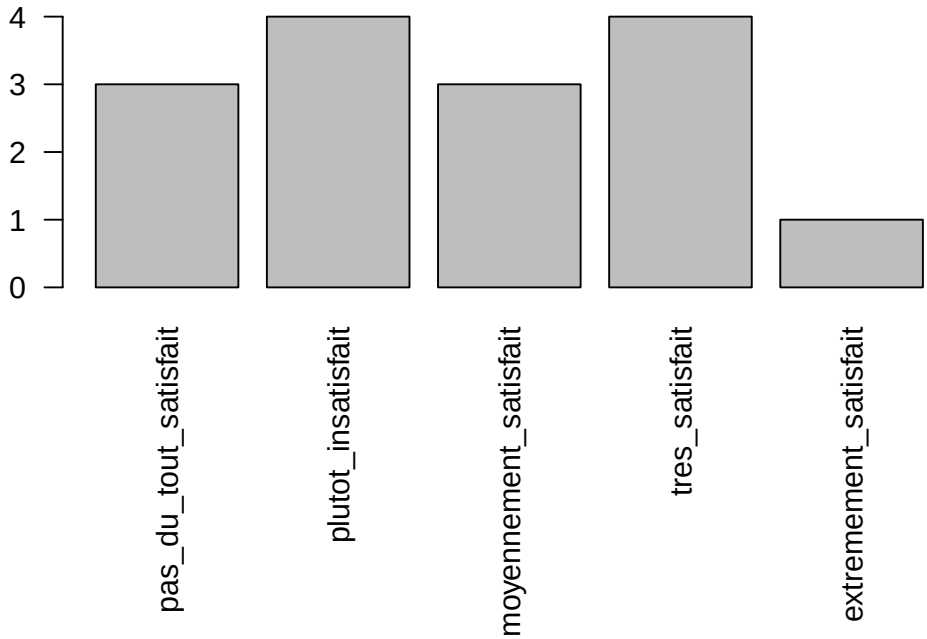
```
barplot(table(df$Sexe))
```



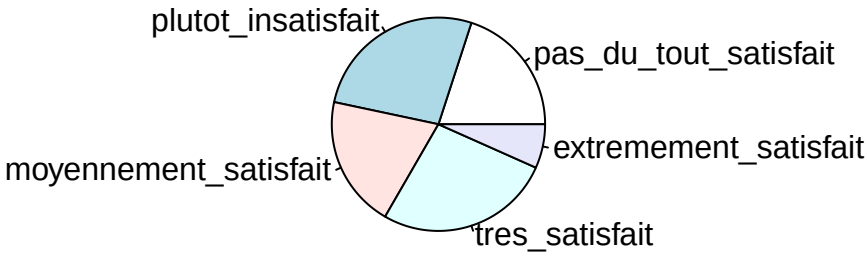

```
tableSatisfaction <- table(df$Satisfaction)
barplot(tableSatisfaction)
```




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




```
pie(tableSatisfaction)
```

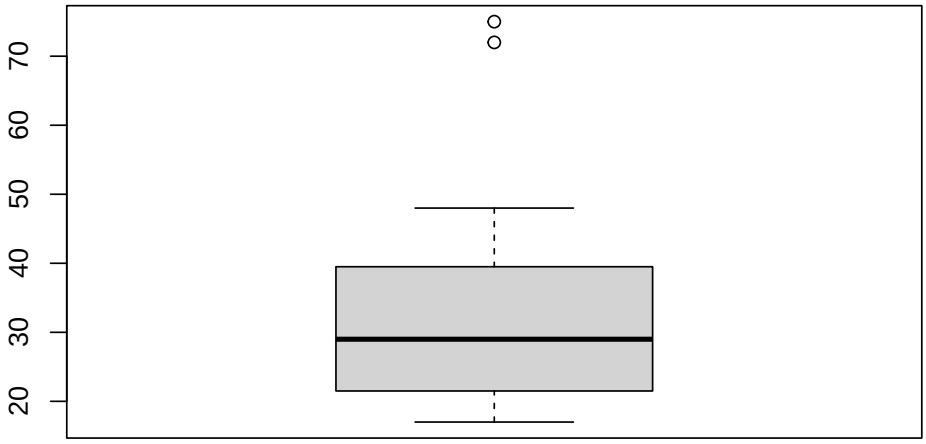


5.2.2 Variables quantitatives

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 nomme un axe.

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


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


20
30
40
50
60
70

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 sommons 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.

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

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

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

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

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

```r
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.

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

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

```r
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:

```r
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 Distribution de Student

La distribution de Student est une distribution de probabilité utilisée en inférence statistique, particulièrement dans le cadre des tests de comparaison de moyennes. Elle est similaire à la distribution normale, mais avec des queues plus larges, ce qui permet de prendre en compte l'incertitude supplémentaire associée à des échantillons réduits. La distribution de Student dépend de ces degrés de liberté. Dans le cadre de ce cours, ces degrés de liberté vous sont

donnés.

Les mêmes fonctions existent pour travailler avec la distribution de Student que celles présentées précédemment pour la distribution normale: `dt()` permet d'obtenir la densité de probabilité pour une valeur donnée, `pt()` permet de calculer la probabilité qu'une variable soit inférieure ou égale à une certaine valeur, `qt()` calcule le quantile pour une probabilité donnée, et `rt()` permet de générer des nombres aléatoires suivant une distribution de Student avec un certain nombre de degrés de liberté. Dans chacune de ces fonctions, il est nécessaire de spécifier l'argument `df` qui correspond au degré de liberté.

Les codes suivants peuvent être utilisés pour calculer ces différentes choses. Prenons une distribution de Student avec 10 degrés de liberté. Le code suivant donne la densité de la distribution de Student pour un valeur de 2.

```
dt(2, df = 10)
```

```
## [1] 0.06114577
```

Le code suivant donne la probabilité que la valeur de la distribution soit inférieure ou égale à 2.

```
pt(2, df = 10)
```

```
## [1] 0.963306
```

Le code suivant donne la probabilité que la valeur de la distribution correspondant au 95e percentile.

```
qt(0.95, df = 10)
```

```
## [1] 1.812461
```

Le code suivant génère un échantillon de 100 valeurs aléatoires suivant une distribution de Student

```
rt(100, df = 10)
```

```
##   [1]  0.491996782  0.692931960  0.223166419  1.399233571  1.948448931
##   [6] -0.974688314  0.019710794 -1.261342778  0.714001690 -0.653644230
##  [11] -0.058149265  0.827884272 -0.593585713 -2.831629628 -2.328379914
##  [16]  1.153533476  0.618555423  0.016367679 -0.439955182  0.688715673
##  [21]  1.020487054  0.259813715  0.402105464 -0.152659020  0.097154234
##  [26] -0.328997352 -0.198634383  1.511699796 -0.176060524  0.471798130
##  [31] -0.078333239 -0.278172052 -0.614644488 -1.548449281  0.421914078
##  [36]  0.554351370  0.171444394  0.359047798  1.639307324 -0.459919848
##  [41] -0.239527010 -1.819500992  2.044075667  0.846471336  0.105014841
##  [46]  0.703689231  0.688088314  0.558887884  1.163575060 -1.069360585
##  [51] -0.312040559 -1.878048891  0.613982680  0.318194489  0.187605622
##  [56]  0.004583574  0.617342907  2.653382612 -0.690964524 -0.365613450
##  [61] -0.977808094 -0.862778073  0.056799455 -0.991031251  0.715344778
```

```
##  [66]  1.306170266  0.262104994  0.368489475  0.412214263 -1.312455869
##  [71]  0.207992240 -1.586964972 -0.354494401 -1.721263676 -0.744498965
##  [76]  2.448878814  0.035364310 -0.863451917  2.551897466 -0.707794335
##  [81] -1.508038971  1.953149173  1.078502213 -1.368140954  0.483839097
##  [86]  0.109863278  0.330925553  1.915406953  1.161535185  0.426198734
##  [91] -1.575039463 -0.152772364  0.029402568 -1.611836402  0.597272592
##  [96]  0.519280797  1.600020826  0.277314039  0.827820451 -0.551902492
```

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 répétitis 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 bouche 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 `i`et 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"
## [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
```

```r
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()`

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

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

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

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

```r
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
```

```r
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 ajouter 0.5 aux notes proposées.

```r
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...

Chapter 8

Les tests statistiques

En statistique, il existe plusieurs tests. Dans le cadre de ce cours, nous allons voir les tests suivants: le test de Student pour un échantillon, pour deux échantillons indépendants, pour des échantillons appariés,...

8.1 Le test de Student pour un échantillon

Le test t de Student pour un échantillon est une méthode d'inférence statistique utilisée pour déterminer si les données d'un échantillon sont significativement différentes d'une valeur hypothétique donnée. Il est utilisé pour comparer deux moyennes. Ce test est basé sur la distribution t de Student et est couramment utilisé lorsqu'on ne connaît pas la variance de la population. Il existe trois types principaux de tests t :

1. Le test bilatéral: tester si la moyenne d'un échantillon est significativement différente d'une valeur donnée
2. Le test unilatéral à droite: tester si la moyenne de l'échantillon est significativement plus grande qu'une valeur donnée
3. Le test unilatéral à gauche: tester si la moyenne de l'échantillon est significativement plus petite qu'une valeur donnée

La fonction `t.test`permet d'effectuer le test de Student. Pour effectuer correctement le test, nous devons remplir trois arguments dans la fonction, à savoir `x`, `alternative`, et `mu` qui correspondent respectivement aux données de l'échantillon, à l'hypothèse alternarive et à la moyenne hypothétique sous l'hypothèse nulle. Précisons que:

1. Le test bilatéral: `alternative="two.sided"`
2. Le test unilatéral à droite: `alternative="greater"`
3. Le test unilatéral à gauche: `alternative="less"`

Afin d'effectuer les tests, commençons par simuler des données issues d'un loi normale.

```
set.seed(123)  # pour la reproductibilité
n <- 30  # taille de l'échantillon
data <- rnorm(n, mean = 55, sd = 10)  # simulation des données
```

Avant de commencer, nous pouvons calculer la moyenne empirique de l'échantillon

```
mean(data)
```

```
## [1] 54.52896
```

Fixons comme hypothèse nulle que la moyenne de l'échantillon est égale à 50: H_0: $\mu_0 = 50$. Dans ce cas, le test est bilatéral et s'effectue selon le code suivant

```
mu_0 <- 50  # moyenne hypothétique sous l'hypothèse nulle
t.test(x = data, mu = mu_0, alternative = "two.sided")
```

```
##
##  One Sample t-test
##
## data:  data
## t = 2.5286, df = 29, p-value = 0.01715
## alternative hypothesis: true mean is not equal to 50
## 95 percent confidence interval:
##  50.86573 58.19219
## sample estimates:
## mean of x
##  54.52896
```

Fixons comme hypothèse nulle que la moyenne de l'échantillon est plus grande que 50: H_0: $\mu_0 > 50$. Dans ce cas, le test est unilatéral à droite et s'effectue selon le code suivant

```
mu_0 <- 50  # moyenne hypothétique sous l'hypothèse nulle
t.test(x = data, mu = mu_0, alternative = "greater")
```

```
##
##  One Sample t-test
##
## data:  data
## t = 2.5286, df = 29, p-value = 0.008576
## alternative hypothesis: true mean is greater than 50
## 95 percent confidence interval:
##  51.48564      Inf
## sample estimates:
## mean of x
##  54.52896
```

Fixons comme hypothèse nulle que la moyenne de l'échantillon est égale à 50: H_0: $\mu_0 < 50$. Dans ce cas, le test est unilatéral à gauche et s'effectue selon le code suivant

```
mu_0 <- 50  # moyenne hypothétique sous l'hypothèse nulle
t.test(x = data, mu = mu_0, alternative = "less")
```

```
##
##  One Sample t-test
##
## data:  data
## t = 2.5286, df = 29, p-value = 0.9914
## alternative hypothesis: true mean is less than 50
## 95 percent confidence interval:
##      -Inf 57.57228
## sample estimates:
## mean of x
##  54.52896
```

L'interprétation du résultat du test se fait comme suit:

1. t: statistique de test
2. df: degrés de liberté
3. p-value: pvaleur à comparer avec le seuil α prédéterminé pour savoir si l'hypothèse nulle est rejetée (p-value$<$ 0.05) ou si l'hypothèse nulle n'est pas rejetée (p-value$>$ 0.05).
4. 95 percent confidence interval: bornes inférieures et supérieures de l'intervalle de confiance

Il est possible d'extraire les informations du test selon le code suivant en utilisant le `$`

```
mu_0 <- 50  # moyenne hypothétique sous l'hypothèse nulle
test <- t.test(x = data, mu = mu_0, alternative = "less")
print(test)
```

```
##
##  One Sample t-test
##
## data:  data
## t = 2.5286, df = 29, p-value = 0.9914
## alternative hypothesis: true mean is less than 50
## 95 percent confidence interval:
##      -Inf 57.57228
## sample estimates:
## mean of x
##  54.52896
```

```r
test$statistic
```

```
##       t
## 2.52858
```

```r
test$p.value
```

```
## [1] 0.9914243
```

```r
test$parameter
```

```
## df
## 29
```

8.2 Le test de Student pour deux échantillons indépendants

Le test de Student pour deux échantillons indépendants (échantillons non appariés) est un test statistique utilisé pour comparer les moyennes de deux groupes indépendants afin de déterminer s'il existe une différence significative entre elles. Il est basé sur l'hypothèse que les données sont issues de distributions normales avec des variances égales ou inégales selon la version du test utilisée. Comme les deux groupes sont indépendants, il est nécessaire de spécifier si la variance entre les deux groupes est similaire ou si elle est différente. Cela se fait grâce à l'argument `var.equal`. Il est nécessaire de remplir l'argument `alternative` pour spécifier si le test est bilatéral, unilatéral à gauche ou unilatéral à droite. De plus, comme deux échantillons de données sont présents, il faut donc remplir les variables `x` et `y`.

Commençons par simuler un exemple avec des variances égales:

```r
set.seed(123) # pour la reproductibilité

groupe1 <- rnorm(30, mean = 100, sd = 15)  # 30 observations, moyenne de 100, écart-ty
groupe2 <- rnorm(30, mean = 110, sd = 15)  # 30 observations, moyenne de 110, écart-ty

t_test <- t.test(x = groupe1, y = groupe2, alternative = "two.sided", var.equal = TRUE

print(t_test)
```

```
##
##  Two Sample t-test
##
## data:  groupe1 and groupe2
## t = -3.7926, df = 58, p-value = 0.0003577
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
```

```
##  -20.444305  -6.318958
## sample estimates:
## mean of x mean of y
##  99.29344 112.67508
```

Nous pouvons également simuler un exemple avec des variances inégales. Nous constatons que le test ci-dessous effectue un test t de Welch qui est l'adaptation du test t de Student lorsque les variances ne sont pas égales.

```
set.seed(123) # pour la reproductibilité

# Génération des données pour deux groupes avec variances inégales
groupe1 <- rnorm(30, mean = 100, sd = 15)  # 30 observations, moyenne de 100, écart-type de 15
groupe2 <- rnorm(30, mean = 110, sd = 25)  # 30 observations, moyenne de 100, écart-type de 25

t.test(x = groupe1, y = groupe2, alternative = "two.sided", var.equal = FALSE)
```

```
##
##  Welch Two Sample t-test
##
## data:  groupe1 and groupe2
## t = -3.2519, df = 52.11, p-value = 0.002013
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -24.522521  -5.807509
## sample estimates:
## mean of x mean of y
##  99.29344 114.45846
```

Par défaut, l'argument `var.equal`est fixé à `FALSE`. Il n'est pas donc pas nécessaire de l'écrire à chaque fois, cependant, nous vous encourageons à le faire de façon à être consitant.

```
t.test(x = groupe1, y = groupe2, alternative = "two.sided")
```

```
##
##  Welch Two Sample t-test
##
## data:  groupe1 and groupe2
## t = -3.2519, df = 52.11, p-value = 0.002013
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -24.522521  -5.807509
## sample estimates:
## mean of x mean of y
##  99.29344 114.45846
```

8.3 Le test de Student pour échantillons appariés

Le test de Student pour échantillons appariés est une autre version du test de Student permettant de comparer deux mesures effectuées sur les mêmes individus. Alors que le test pour deux échantillons indépendants permet de comparer deux groupes différents, comme les hommes et les femmes, ce test permet de comparer un échantillon dans deux conditions ou dans deux temps de mesure. Il est utilisé lorsqu'on veut déterminer si la différence moyenne entre deux conditions est significativement différente de zéro. Ce test est également très utile dans des cas tels que l'évaluation de l'effet d'un traitement avant/après sur un même groupe de sujets.

Comme deux échantillons de données sont présents, il faut donc remplir les variables `x` et `y`. De plus, comme les échantillons sont appariés, l'argument `paired = TRUE` doit être spécifié. Il est toujours nécessaire de spécifier l'argument `alternative`pour l'hypothèse alternative.

```r
# simuler et organiser les données
set.seed(123)
n <- 30
t1 <- rnorm(n, mean = 50, sd = 10)
t2 <- t1 + rnorm(n, mean = 10, sd = 5)

df <- data.frame(
  individu = 1:n,
  avant = t1,
  apres = t2
)

#test
t.test(x = df$avant,  y = df$apres, alternative = "two.sided", paired = TRUE)
```

```
##
##  Paired t-test
##
## data:  df$avant and df$apres
## t = -14.287, df = 29, p-value = 1.172e-14
## alternative hypothesis: true mean difference is not equal to 0
## 95 percent confidence interval:
##  -12.450901  -9.332482
## sample estimates:
## mean difference
##       -10.89169
```

8.4 La puissance d'un test de Student

La puissance statistique d'un test est la probabilité de détecter un effet réel lorsqu'il existe, c'est-à-dire de rejeter correctement l'hypothèse nulle. Une puissance élevée, généralement fixée à 80 % ou plus, est essentielle pour minimiser le risque d'erreur de type II, à savoir ne pas détecter un effet réel et garantir que les résultats d'une étude sont fiables et interprétables.

La fonction `power.t.test`permet de calculer la puissance d'un test ou de déterminer des paramètres pour obtenir une puissance fixée. Les arguments à spécifier sont les suivants:

1. `n`: le nombre d'observation par groupe
2. `delta`: la différence attendue entre les moyennes
3. `sd`: l'écart-type des données dans chaque groupe
4. `sig.level`: le seuil α, donc le risque d'erreur de type I
5. `power`: la puissance souhaitée
6. `type`: le type de test donc "one.sample" pour un échantillon indépendant, "two.sample" pour deux échantillons indépendants, "paired" pour des échantillons appariés
7. `alternative`: l'hypothèse alternative simplifiée, donc "two.sided" pour un test bilatéral, "one.sided" pour un test unilatéral.

Si l'argument `power` n'est pas spécifié, alors la fonction retourne la puissance du test. Si l'argument `n` n'est pas spécifié, alors la fonction retourne la taille de l'échantillon nécessaire pour détecter la différence souhaitée avec les paramètres spécifiés. Les exemples suivants permettent de comprendre comment travailler avec les arguments:

On veut déterminer combien de participants sont nécessaires pour détecter une différence moyenne de 5 points avec un écart-type de 10, une puissance de 80% et un seuil alpha de 0.05.

```
power.t.test(delta = 5, sd = 10, power = 0.8, sig.level = 0.05, type = "one.sample")
```

```
##
##      One-sample t test power calculation
##
##               n = 33.3672
##           delta = 5
##              sd = 10
##       sig.level = 0.05
##           power = 0.8
##     alternative = two.sided
```

On veut déterminer la taille d'échantillon nécessaire pour comparer deux groupes avec une différence moyenne attendue de 5, un écart-type de 10 et une puissance de 80%. Le test compare deux groupes indépendants, donc l'échantillon total sera deux fois la taille renvoyée.

```r
power.t.test(delta = 5, sd = 10, power = 0.8, sig.level = 0.05, type = "two.sample")
```

```
##
##      Two-sample t test power calculation
##
##               n = 63.76576
##           delta = 5
##              sd = 10
##       sig.level = 0.05
##           power = 0.8
##     alternative = two.sided
##
## NOTE: n is number in *each* group
```

On veut calculer la puissance d'un test apparié avec 30 participants, une différence attendue de 3 et un écart-type de 8.

```r
power.t.test(n = 30, delta = 3, sd = 8, sig.level = 0.05, type = "paired")
```

```
##
##      Paired t test power calculation
##
##               n = 30
##           delta = 3
##              sd = 8
##       sig.level = 0.05
##           power = 0.5102817
##     alternative = two.sided
##
## NOTE: n is number of *pairs*, sd is std.dev. of *differences* within pairs
```

Chapter 9

Régression linéaire simple

Le modèle de régression linéaire simple cherche à déterminer la relation entre **une** variable indépendante (x) et une variable dépendante (y). Plus précisément, ce modèle cherche à déterminer si la variable indépendante (qui peut être qualitative ou quantitative) prédit la variabilité de la variable dépendante (toujours quantitative continue).

Par exemple:

- Est-ce que le nombre d'heures supp. prédit le stress au travail?

  `stress ~ heures supp`

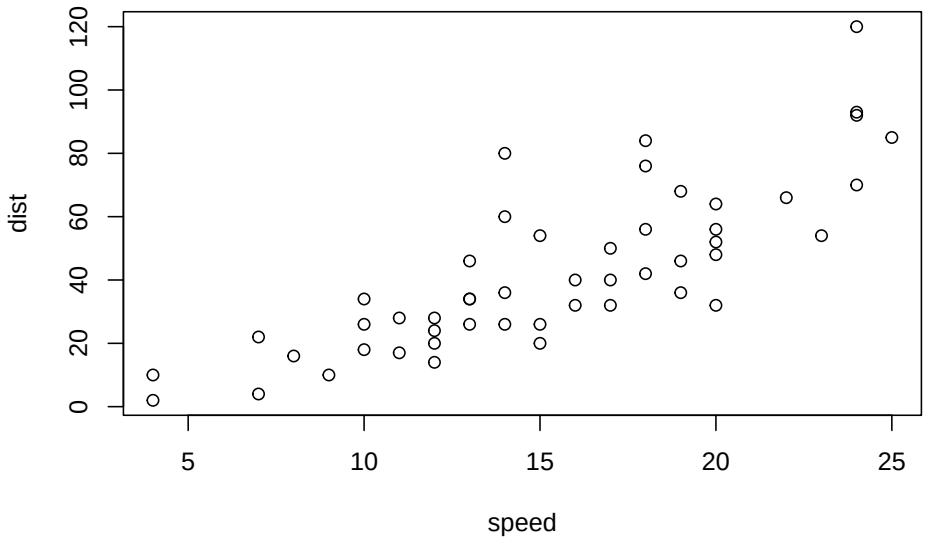
- Est-ce que la taille prédit le poids?

  `poids ~ taille`

9.1 Visualisation du lien linéaire entre deux variables

Afin de visualiser la nature du lien entre deux variables continues et vérifier si ce lien est linéaire, nous vous proposons de construire un diagramme de dispersion. Cette représentation graphique a deux axes: la variable indépendante sur l'axe des x et la variable dépendante sur l'axe des y.

```
# Par exemple, le diagramme de dispersion entre la vitesse d'une voiture (x = speed) et sa distan
plot(dist ~ speed, data = cars)
```

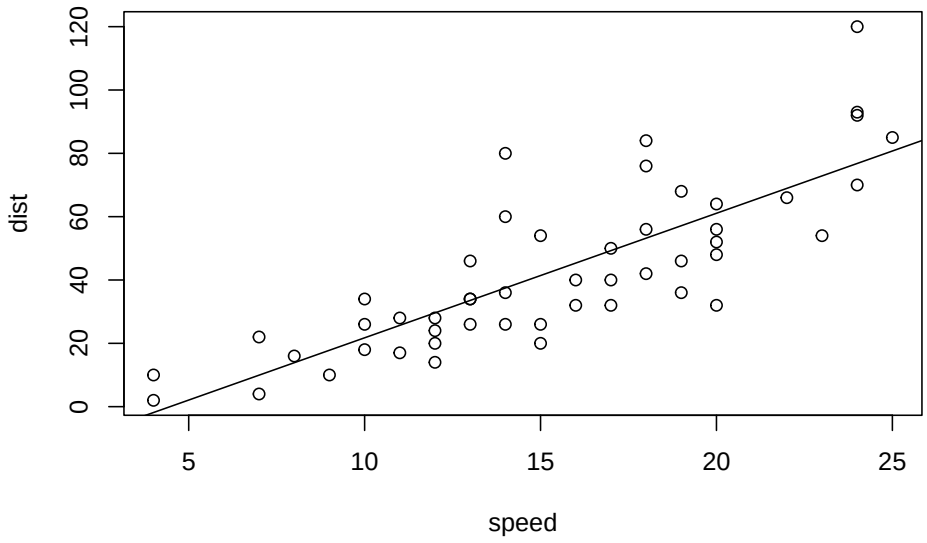


Les modèles de régression linéaire vont calculer et estimer la meilleure droite qui passe au milieu des données de telle sorte que cette droite représente la relation entre les deux variables. La fonction *lm( )* dans R calcule la pente et l'intercept de cette droite.

```
m1 <- lm(dist ~ speed, data = cars)
```

Le code suivant permet, une fois le modèle de régression calculé et stocké dans un objet (par exemple m1), de dessiner la droite de régression sur le diagrame de dispersion. Attention, il faut pour cela lancer les deux lignes de code en même temps.

```
# pour notre exemple :
plot(dist~speed,data=cars)
abline(m1)
```



9.2 Conditions d'application

Avant de pouvoir modéliser nos données par une régression linéaire, quatre conditions d'applications doivent être respectées. Il s'agit de prérequis théoriques et mathématiques:

1. La relation entre les variables doit être linéaire.

2. Les données doivent être indépendantes les unes des autres.

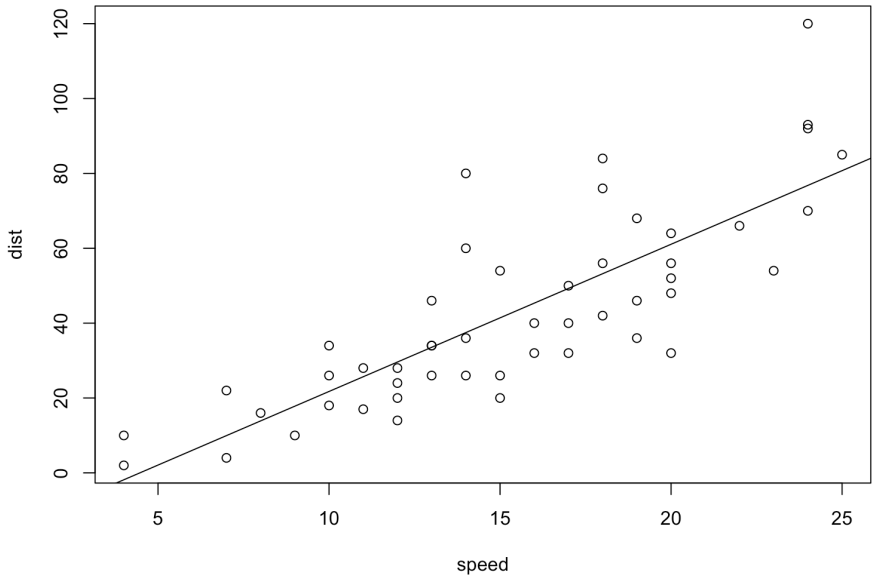
3. Les résidus doivent être distribués normalement, avec une moyenne de 0.

4. La variance doit être homogène (homoscedasticité). En d'autres mots, l'erreur doit être homogène pour toutes les valeurs de X.

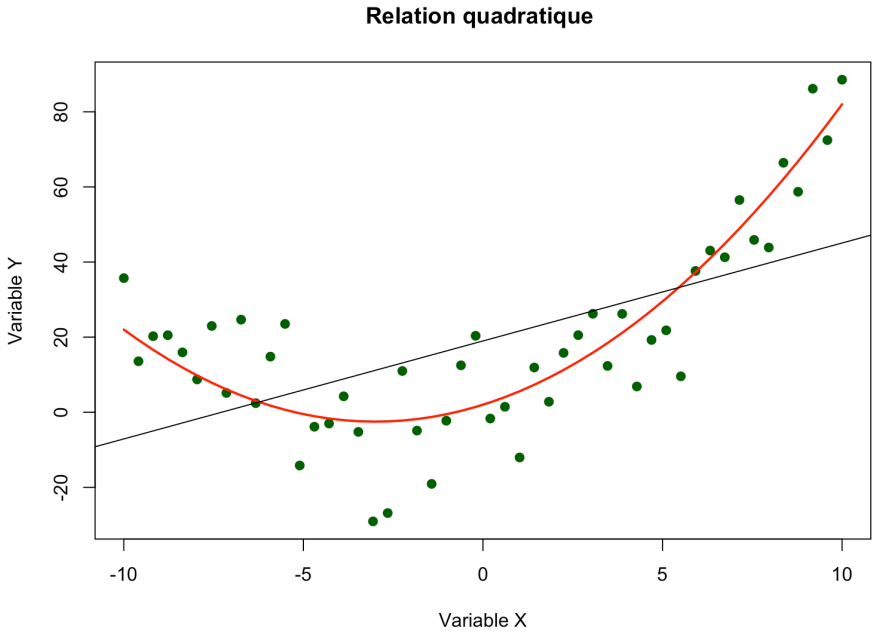
9.2.1 Relation linéaire

Afin de vérifier le lien linéaire entre deux variables, nous allons utiliser une méthode graphique: la visualisation des données. Comme vu au point 9.1, nous allons utiliser la fonction *plot( ).* Le but ensuite est de juger si le lien entre les données pourrait être correctement estimé par une droite.

Exemple 1 : Relation linéaire



Exemple 2 : Relation non-linéaire (quadratique)



9.2.2 Données indépendantes

Afin de pouvoir modéliser nos données par une régression linéaire, il est important que ces dernières soient indépendantes les unes des autres. Pour vérifier l'application de cette condition, il est important de connaître les conditions de récolte des données, autrement dit, le protocol expérimental.

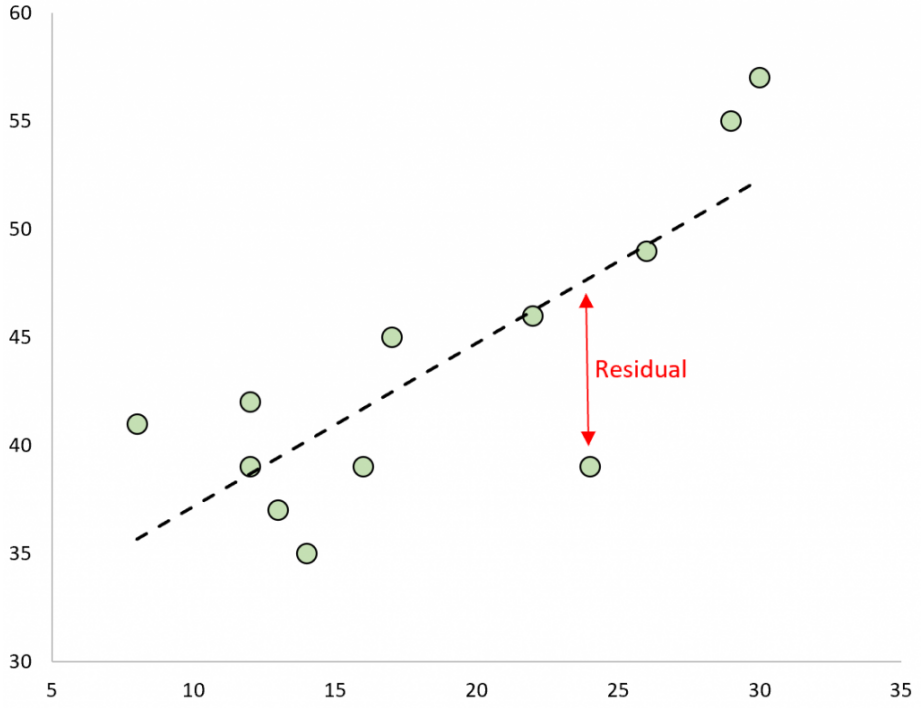
Par exemple, s'il s'agit de données pairées, vous n'allez pas pouvoir modéliser vos données par une régression linéaire simple.

Dans le cas des exercices que nous vous donnons à faire, vous pouvez, sauf indication contraire, estimer que les données pertinentes sont bel et bien indépendantes.

9.2.3 Résidus distribués normalement

Les deux dernières conditions d'application concernent les résidus/erreurs de votre modèle de régression linéaire. Ces résidus sont les écarts entre les valeurs observées et les valeurs prédites par votre modèle. En d'autres mots, les résidus représentent l'écart entre les données récoltées et la droite de régression définie par le modèle. Vous pouvez aussi comprendre les résidus comme une représentation du 'bruit' que votre régression n'arrive pas à modéliser.

Par exemple, la représentation, en rouge, du résidu (ou residual) d'une des observation d'un modèle de régression linéaire.



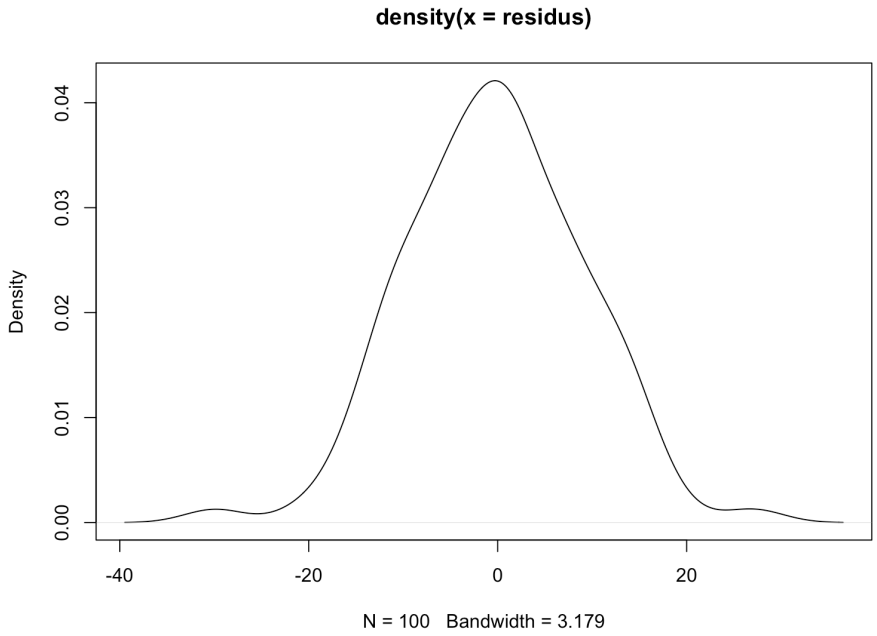
Ensuite, afin de pouvoir mesurer et observer les résidus, il faut d'abord définir votre modèle de régression à l'aide de la fonction *lm( )*. Ensuite, afin de connaître les valeurs des résidus, nous vous proposons d'utiliser la fonction *resid( )*.

```
modele_1 <- lm(dist ~ speed, data = cars) # définition du modèle de régression linéair
residus <- resid(modele_1) # calcul des résidus
```

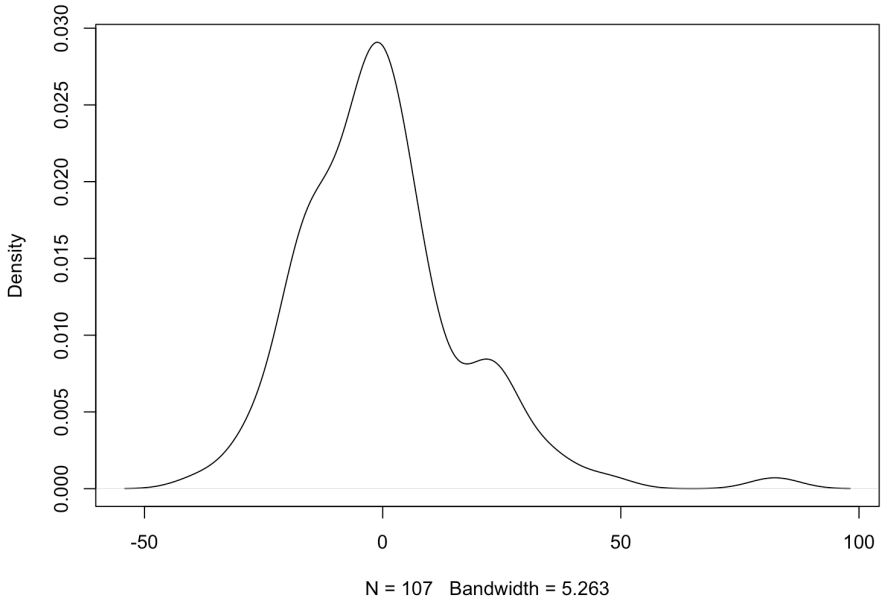
Afin de vérifier la normalité de la distribution des erreurs/résidus, trois options s'offrent à vous.

1. Vous pouvez visualiser la distribution des résidus par un graphique de densité. **Critère de décision**: est-ce que cette distribution ressemble à une distribution normale centrée en 0?

```
# exemple de résidus distribués normalement
plot(density(residus))
```



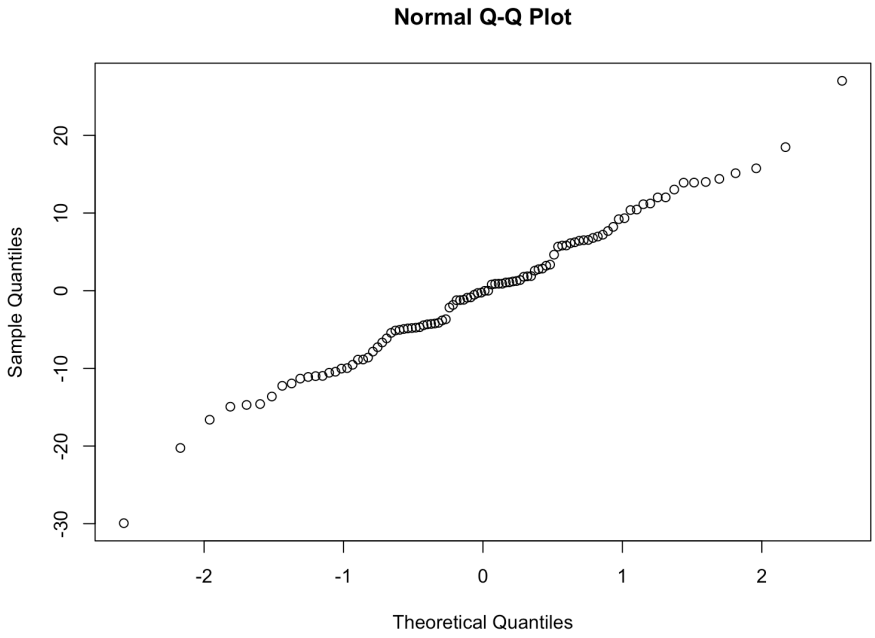

```
# exemple de résidus qui ne sont pas tout à fait distribués normalement
plot(density(residus2))
```



2. Vous pouvez également visualiser la normalité de la distribution des résidus

par un QQplot. Un QQplot représente les résidus (sur l'axe des y) en fonction des valeurs théoriques des résidus si la distribution était parfaitement normale. **Critère de décision**: Est-ce que les points sont sur la diagonale.

```
# mêmes exemples que ci-dessus, résidus distribués normalement
qqnorm(residus)
```



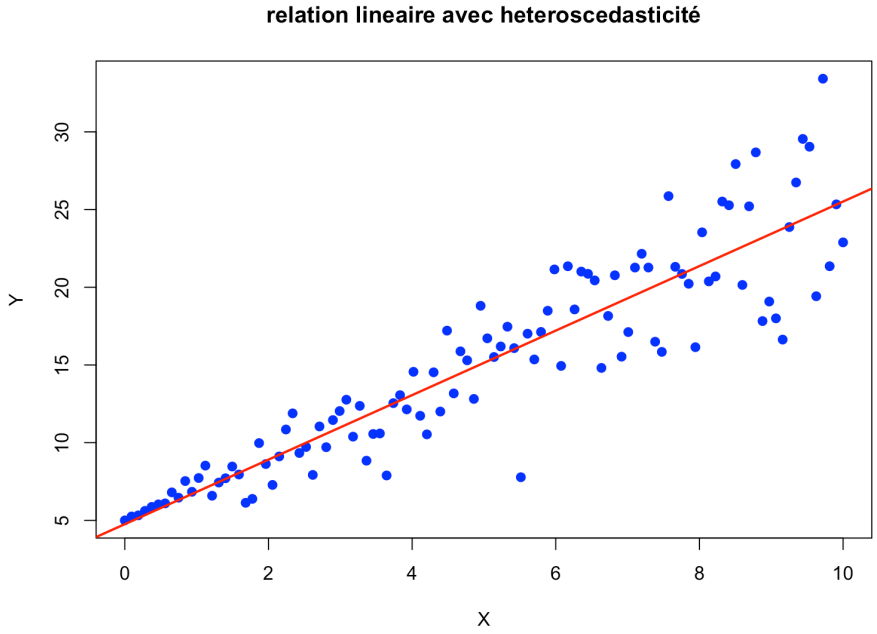
3. Finalement, de manière intuitive, le diagramme de dispersion avec la droite de régression vous permet également d'avoir une idée de le distribution des résidus. La condition de normalité est respectée si il y a autant de résidus en haut et en bas de la droite, et que les résidus ne sont pas plus éloignés de la droite en haut qu'en bas (ou vice-versa)

9.2.4 Homoscedasticité

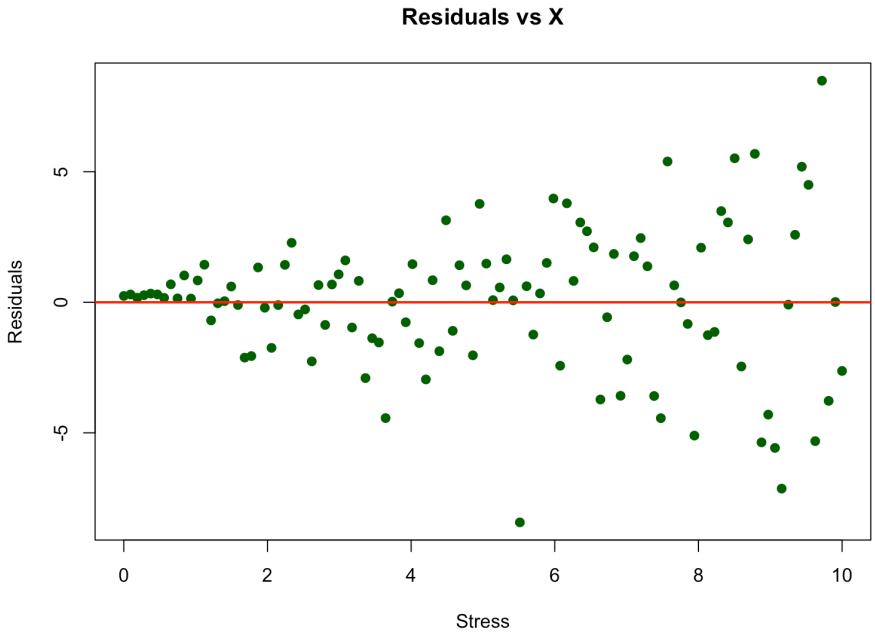
Finalement, il est essentiel que les résidus se distribuent de manière homogène pour toutes les valeurs de x. En d'autres termes, que la précision de la droite de régression reste la même pour toutes les valeurs de x. Afin de vérifier cette condition, vous allez visualer les données autour de la droite de régression et faire un graphique des résidus en fonction de votre variable indépendante (x). **Critère de décision:** est-ce que les résidus pourraient être contenus dans un rectangle (homoscédasticité) ou non (hétéroscédasticité).

Exemple 1 : Relation linéaire avec hétéroscedasticité (variance

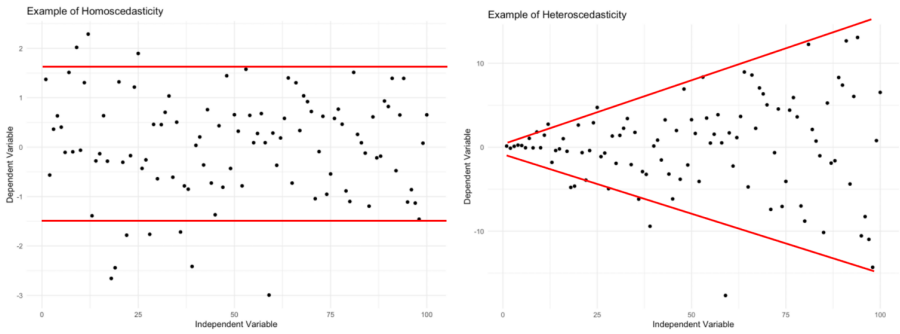
hétérogène)




```r
plot(Stress, residuals) # graphique des résidus en fonction de x
abline(h = 0, col = "red", lwd = 2)
```



Exemple 2 : Comparaison entre homoscédasticité et hétéroscédasticité



A nouveau, simplement regarder le diagramme de dispersion et la droite de régression vous permet également d'avoir une idée de si cette conditoin est respectée. Si elle l'est, les résidus sont similaires tout au long de la droite de régression.

9.3 Paramètres et tests d'hypothèses

Si toutes les conditions d'application sont respectées, vous allez pouvoir interprêter les résultats de votre modèle de régression linéaire simple.

En guise de rappel, pour spécifier votre modèle de régression vous allez utiliser la fonction *lm( )*. Ensuite, pour en observer les paramètres/résultats, vous allez utiliser la fonction *summary( )*.

```
modele_1 <- lm(dist ~ speed, data = cars)
summary(modele_1)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)                    1
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123 *
## speed         3.9324     0.4155   9.464 1.49e-12 ***
## ---             2          3       4         5
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom    6
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438   7
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12      8
```

En quelques mots, voici ce que représentent les différents indices et paramètres de votre sortie. Vous avez deux blocs principaux, un premier qui vous informe sur les paramètres du modèle (2:5), un second qui vous informe sur la validité du modèle (6:8).

Bloc 1 - Les paramètres de votre modèle

1. Le rappel de votre équation de régression linéaire
2. Les valeurs de paramètres de votre modèle. Vous avez une valeur pour l'ordonnée à l'origine (intercept) ainsi que pour la variable indépendante (ou prédicteur)
3. L'erreur standard de chacun des paramètres
4. Un score t est calculé pour chaque paramètre afin d'en tester la significativité
5. La p-valeur de chaque paramètre. Cette p-valeur vous permet de juger de la significativité de vos différents paramètres

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -17.5791     6.7584  -2.601   0.0123 *
speed         3.9324     0.4155   9.464 1.49e-12 ***
---
```

2 3 4 5

Bloc 2 - La validité de votre modèle

6. L'erreur standard des résidus du modèle
7. Les coefficients de détermination de votre modèle (R-carré et R-carré ajusté). Ces derniers vous indiquent la part de variance de votre variable dépendante qui est expliquée par votre modèle.
8. Cette F-statistic et sa probabilité critique vous permettent de conclure sur la significativité de votre modèle. Est-ce que ce dernier 'prédit' significativement votre variable dépendante? Plus concrètement, ces valeurs permettent de rejeter ou non l'hypothèse nulle suivante: le modèle avec la variable indépendante et un modèle avec uniquement un intercept ne diffèrent pas quant à leur capacité à expliquer la variabilité de la variable dépendante.

```
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

6 7 8

9.4 Standardiser une variable

Souvent, afin de faciliter l'interprétation et la comparaison des paramètres, nous allons standardiser nos variables avant de les insérer dans notre modèle de régression linéaire.

Le processus de standardiser une variable équivaut à:

- centrer la variable - soustraire la moyenne de la variable à chaque score individuel
- réduire la variable - diviser chaque score par l'écart-type de la variable

En d'autre mot, standardiser une variable est le même procédé qu'en calculer le score z.

```
speed_z <- (speed - mean(speed)) / sd(speed) #vitesse standardisée
dist_z <- (dist - mean(dist)) / sd(dist) #distance de freinage standardisée
```

```r
modele_z <- lm(dist_z ~ speed_z) #modèle sur variable standardisées
summary(modele_z)
```

```
##
## Call:
## lm(formula = dist_z ~ speed_z)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.12805 -0.36964 -0.08816  0.35758  1.67646
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 4.052e-17  8.440e-02   0.000        1
## speed_z     8.069e-01  8.526e-02   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5968 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```