An Introduction to R

Vittorio Perduca (Université de Paris)

September 2020

Contents

1 Welcome **5**

2 Introduction **7**

2.1 How to use this document 8

2.2 Useful references 8

2.3 Installing R and Rstudio 8

2.4 Getting started with RStudio 8

2.5 Packages 9

3 Language basics **11**

3.1 Assignments, arithmetic operations 11

3.2 Mode, length and class 12

4 Vectors, matrices and functions **17**

4.1 Vectors 17

4.2 Matrices 20

4.3 Operations on numerical vectors and matrices 21

4.4 Factors 24

4.5 User-defined functions 25

4.6 Exercises 25

5 Lists and data frames **27**

5.1 Lists 27

5.2 Data frames 28

5.3 Importing and exporting data 30
5.4 Exercise 32

6 Descriptive statistics **33**
6.1 Data used 33
6.2 Univariate analysis 34
6.3 Bivariate analysis 40

7 Base graphics **45**
7.1 Basic constructions and scatter plots 45
7.2 Line graphs 50
7.3 Graphical parameters 52
7.4 Histograms 54
7.5 Exercises 56

8 Probability distributions **59**
8.1 Discrete distributions of discrete random variables 60
8.2 Continuous distributions 60
8.3 Example: the Law of Large Numbers 61

Chapter 1

Welcome

This document is a tutorial for self-learning the basic use of `R`. It was prepared for the students enrolled in the first year of the AIRE Life Sciences Master Program of the Centre de Recherche Interdisciplinaire at the Université de Paris.

This document was written with the `R` package `bookdown` (Xie, 2020), (Xie, 2016) and is under continuous development, please report any issues to `vittorio.perduca at u-paris.fr`.

Chapter 2

Introduction

R is a statistical language developed from the 1990s at the University of Auckland, New Zealand. Its main implementation is the open source software (free and modifiable) R. R is normally used from the graphical user interface (GUI) and development interface RStudio.

R is an interpreted language (likePython) that we use from the command line:

```
print ('Hello world!')
```

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

Alternatively, you can run a script, that is a series of commands found in a file with the extension .R.

The R user community is very active in the scientific world (statistics, data science, bioinformatics, social sciences, ...) and more and more in companies. One of the advantages of R is the wealth of *packages* developed by users and developers that can be installed to increase its capabilities in many areas of statistics.

In addition, R has a very complete documentation. You can access help by typing ? followed by the *function* you are interested in:

```
? rnorm
```

You can also find a lot of information on the web: by searching on Google for a problem related to an R task, you almost always find an answer (often in threads opened on Cross Validated).

2.1 How to use this document

Although basic objects and commands are covered, this introduction is not a complete reference to the `R` language, so you will have to use the help and search for information on the web. Before moving on to the exercises at the end of each chapter, you are encouraged to type the commands and understand the result: the best way to learn a language is to write and debug lots of lines of code!

2.2 Useful references

A complete reference to `R` is the official introduction that can be found on the CRAN website: https://cran.r-project.org/doc/manuals/R-intro.html. For French readers, an excellent reference is Vincent Goulet's book *Introduction à la programmation R*, which can be downloaded at https://cran.r-project.org/doc/contrib/Goulet_introduction_programmation_R.pdf

2.3 Installing R and Rstudio

1. Download the `R` distribution that is appropriate for your machine from https://cran.r-project.org/ and install it by double-clicking on the installation file.
2. Dowanload and install RStudio Desktop from https://rstudio.com/products/rstudio/download/.

2.4 Getting started with RStudio

The RStudio workspace is generally divided into four pans: starting from the top left and going clockwise we find:

1. a script editor
2. the environment listing the variables in memory and the history of commands that were executed
3. a window for graphical outputs, the help, and the file and package managers.
4. the `R` console, ie the command line.

Typically we type the code in the editor and then run it in the console. To do this we place the cursor in the line we want to execute and we send the command to the console using the combination `cmd Return` in macOS and`ctrl Return` under Linux and Windows.

2.5 Packages

To install a package from the console:

```
install.packages("dplyr")
# download and install the dplyr package used for data manipulation
```

Once a package is installed, it will have to be loaded into memory each time a new session is opened:

```
library(dplyr) # quotes are not needed here
# require(dplyr) # equivalent function
```

Chapter 3

Language basics

3.1 Assignments, arithmetic operations

There are two types of R commands: expressions and assignments.

Expression

```
cos(pi)
```

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

Assignments and expressions

```
x <- 1 + 2 # or x = 1 + 2
x
```

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

```
y = 4
x == y
```

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

Using ; we can type two commands on the same line before executing them:

```
e <- exp (1); log (e)
```

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

Some examples of arithmetic and Boolean operators:

```
3*4
```

```
## [1] 12
```

```
12/3
```

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

```
2^3
```

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

```
sqrt (16)
```

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

```
1 == 2
```

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

```
1!= 1
```

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

```
FALSE & TRUE # and
```

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

```
FALSE | TRUE # or
```

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

3.2 Mode, length and class

In R, everything is an *object*. The *mode* specifies what an object can contain. The main modes are:

- `numeric`: real numbers
- `character`: character strings
- `logical`: logical values true / false
- `list`: list, collection of objects
- `function`: function

`numeric`,`character`, and `logical` objects are *simple* objects that can contain data of only one type. On the contrary, `list` mobjects are special objects that can contain other objects.

You can access the mode of an object with the `mode ()` function:

```
age = c(33,28, 33) # The concatenation function c() allows to create vectors
mode(age)
```

```
## [1] "numeric"
```

```
names <- c('Daniel', 'Jehanne', 'Romain')
mode(names)
```

```
## [1] "character"
```

```
my.list <- list(Names = names, Age = age)
mode(my.list)
```

```
## [1] "list"
```

```
mode(is.integer(pi))
```

```
## [1] "logical"
```

```
mode
```

```
## function (x)
## {
##     if (is.expression(x))
##         return("expression")
##     if (is.call(x))
##         return(switch(deparse(x[[1L]])[1L], `(` = "(", "call"))
##     if (is.name(x))
##         "name"
##     else switch(tx <- typeof(x), double = , integer = "numeric",
##         closure = , builtin = , special = "function", tx)
## }
## <bytecode: 0x7fae3e7a0e90>
## <environment: namespace:base>
```

Besides the mode, an object also has a *length*, defined as the number of elements it contains:

```
length(age)
```

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

```
length(names)
```

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

```
length(my.list)
```

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

The *class* of an object specifies its behavior and therefore its way of interacting with operations and functions. An important example are *data frames*: special lists whose elements all have the same length. The class of a data frame is different from that of generic lists and data frames have an indexing system that does not exist for other lists:

```
class(my.list)
```

```
## [1] "list"
```

```
my.data.frame = data.frame(names, age)
mode(my.data.frame)
```

```
## [1] "list"
```

```
class(my.data.frame)
```

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

```
my.data.frame[1,2] # to extract the 1st element from the 2nd "column"
```

```
## [1] 33
```

```
# Try the following command:
# my.list[1,2]
```

A special object is the missing value `NA`. By default, its mode is `logical`, however`NA` is neither `TRUE` nor`FALSE`. To test if a value is missing we will use the `is.na ()` function:

```
NA == NA # not what we want
```

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

```
is.na(NA)
```

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

```
is.na(mean(c(1,4, NA)))
```

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

Chapter 4

Vectors, matrices and functions

4.1 Vectors

The R basic object is the *vector* (a scalar is considered as a vector of length one). The most used function to create a vector is the concatenation:

```
price <- c(150, 162, 155, 157); price
```

```
## [1] 150 162 155 157
```

Indexing is done through brackets:

```
price[1] # Unlike in Python, the first index is always 1!!
```

```
## [1] 150
```

```
price[c(1.3)]
```

```
## [1] 150
```

```
price[-(1:2)] # to extract all elements except the 1st and 2nd
```

```
## [1] 155 157
```

One can also use a boolean indexing vector, the extracted elements are obviously those corresponding to the `TRUE` values. For example to extract prices greater than 156:

```
price > 156 # the boolean vector
```

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

```
price[price > 156]
```

```
## [1] 162 157
```

An alternative is given by the `which ()` function which returns the indices whose elements satisfy a logical condition:

```
which(price > 155)
```

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

```
price[which(price > 156)]
```

```
## [1] 162 157
```

You can use the indexing to change an element:

```
price [1] <- 0; price
```

```
## [1]   0 162 155 157
```

It is possible to give labels to the elements of a vector and extract elements based on them:

```
names(price)
```

```
## NULL
```

```
# NULL is a special object with NULL mode that reads "no container"
names(price) <- c('model.1', 'model.2', 'model.3', 'model.4')
price
```

```
## model.1 model.2 model.3 model.4
##       0     162     155     157
```

```
price['model.3']
```

```
## model.3
##     155
```

In a vector, all the elements must have the same mode:

```
x <- c(1,2, 'a', 'b'); x
```

```
## [1] "1" "2" "a" "b"
```

```
mode(x)
```

```
## [1] "character"
```

To generate the vector of the first n integers we use the syntax `1:n`

```
1:10
```

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

```
2:6
```

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

To generate more general sequences we use the `seq()` function:

```
seq(from = 2, to = 20, by = 2) # or more simply seq(2,20,2)
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

We can create a vector of repeated elements with `rep()`:

```
rep(1, len = 3) # same thing as rep (1,3)
```

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

```
rep(NA, 4)
```

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

4.2 Matrices

A matrix is a vector with a `dim` attribute of length two. All the elements of a matrix therefore have the same mode. To create a matrix:

```
M <- matrix(2:7, nrow = 2, ncol = 3); M
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    3    5    7
```

```
matrix (2:7, nrow = 2, ncol = 3, byrow = TRUE)
```

```
##      [,1] [,2] [,3]
## [1,]    2    3    4
## [2,]    5    6    7
```

By default `matrix ()` fills the new matrix one column after another. Indexing is done through brackets:

```
M[2,] # 2nd line
```

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

```
M[, 3] # 3rd column
```

```
## [1] 6 7
```

```
M[2.3]
```

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

```
M[3]
```

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

```
M[, -2] # to extract all columns except the 2nd
```

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

To vertically (resp. horizontally) merge two matrices we use `rbind()` (resp. `cbind()`):

```
cbind (M, -M)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    2    4    6   -2   -4   -6
## [2,]    3    5    7   -3   -5   -7
```

```
rbind(M, 2 * M)
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    3    5    7
## [3,]    4    8   12
## [4,]    6   10   14
```

4.3 Operations on numerical vectors and matrices

Element wise operations:

```
v <- c(3,4,1,6)
v + 2
```

```
## [1] 5 6 3 8
```

```
v * 2
```

```
## [1]  6  8  2 12
```

```
v * v
```

```
## [1]  9 16  1 36
```

```
v / 2
```

```
## [1] 1.5 2.0 0.5 3.0
```

```
v / v
```

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

```
v + v^2
```

```
## [1] 12 20  2 42
```

```
sqrt(M)
```

```
##          [,1]     [,2]     [,3]
## [1,] 1.414214 2.000000 2.449490
## [2,] 1.732051 2.236068 2.645751
```

```
M * M
```

```
##      [,1] [,2] [,3]
## [1,]    4   16   36
## [2,]    9   25   49
```

```
# Try the following command:
# M + v
```

Transpose, multiplication, inverse:

```
t(M)
```

```
##      [,1] [,2]
## [1,]    2    3
## [2,]    4    5
## [3,]    6    7
```

```
N <- M[, -3]
N %*% diag(1,2) # row by column product matrix
```

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

```
# diag (1,2) builds the 2x2 diagonal matrix where all the
# diagonal elements are equal to 1, ie the 2x2 identity matrix
solve(N)
```

```
##      [,1] [,2]
## [1,] -2.5    2
## [2,]  1.5   -1
```

```
solve(N) %*% N # checking if solve(N) is the inverse of N
```

```
##      [,1]          [,2]
## [1,]    1 1.776357e-15
## [2,]    0 1.000000e+00
```

The transpose of a vector is a row matrix:

```
V <- t(v)
dim(V)
```

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

```
t(V)
```

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

Pay attention to the following examples:

```
v %*% t(v) # v is considered a column vector!
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    9   12    3   18
## [2,]   12   16    4   24
## [3,]    3    4    1    6
## [4,]   18   24    6   36
```

```
t(v) %*% v # ditto
```

```
##      [,1]
## [1,]   62
```

```
diag(1,4) %*% v # ditto
```

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

```
v %*% v # v is both considered a row-vector and a column-vector
```

```
##      [,1]
## [1,]   62
```

4.4 Factors

A factor is a vector used to represent qualitative variables, ie a variable with discrete values. Its values, or categories, are called the `levels` in `R`.

```
city <- c('paris', 'lyon', 'lyon', 'paris', 'nantes')
fact.city <- as.factor(city); fact.city
```

```
## [1] paris  lyon   lyon   paris  nantes
## Levels: lyon nantes paris
```

```
class(fact.city)
```

```
## [1] "factor"
```

```
levels(fact.city)
```

```
## [1] "lyon"   "nantes" "paris"
```

A factor has the `numeric` mode. The reason for this counter-intuitive fact is that the elements of a factor are represented as integers corresponding to the lexicographic order of their values:

```
mode(fact.city)
```

```
## [1] "numeric"
```

```
as.numeric(fact.city)
```

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

4.5 User-defined functions

Example:

```
my.function <- function(x, y = 10) {# the default value of y is 10
  z = x-y
  return(z)
}
my.function(2)
```

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

```
my.function(2,4)
```

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

```
my.function(y = 1, x = 4)
```

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

Any variable defined in a function is *local* and does not appear in the workspace: try to run

```
z
```

4.6 Exercises

1. Let x be a vector with the elements of a sample:

```
##  [1] 45 63 17 32 54 57 41 29 34 37 18 39 46 43
```

- Write a code to give
  - the third element of the sample
  - the four elements of the sample
  - the items strictly greater than 35.
  - all elements except those in positions 3, 9 and 12.
- Replace the first element by a missing value and give the position of all elements less than 30.

2. Write a function `weighted_average` that takes as inputs two vectors $x = (x_1, \dots, x_n)$ and $w = (w_1, \dots, w_n)$ and computes the weighted mean

$$\frac{1}{\sum_{i=1}^{n} w_i} \sum_{i=1}^{n} w_i x_i$$

Chapter 5

Lists and data frames

5.1 Lists

Lists are special vectors that can store elements of any mode (including other lists).

```
age = c(33,28, 33)
names <- c('Daniel', 'Jehanne', 'Romain')
my.list <- list(Names = names, Age = age)
```

Like any other vector, a list is indexed by the `[...]` operator, however, note that the result will be a list containing as unique element the desired item:

```
my.list[1]
```

```
## $Names
## [1] "Daniel"  "Jehanne" "Romain"
```

```
mode(my.list[1])
```

```
## [1] "list"
```

To get the desired item directly, we therefore use the `[[...]]` operator or the `$` operator followed by the name of the element (if available):

```
my.list[[1]]
```

```
## [1] "Daniel"  "Jehanne" "Romain"
```

```
my.list$age
```

```
## NULL
```

The elements of a list can have different lengths:

```
city <- c('paris', 'lyon', 'lyon', 'paris', 'nantes')
my.list$city <- city
my.list
```

```
## $Names
## [1] "Daniel"  "Jehanne" "Romain"
##
## $Age
## [1] 33 28 33
##
## $city
## [1] "paris"  "lyon"   "lyon"   "paris"  "nantes"
```

5.2 Data frames

The most widely data containers is the data frame, a special list of class `data.frame` in which all elements have the same length. For this reason, a data frame is represented in the form of a two-dimensional array whose columns are its elements. Typically, in a data frame the columns represent the **variables** and the rows the **observations**. Unlike matrices, the elements of a data frame can have different modes.

```
id <- c('id.453', 'id.452', 'id.455', 'id.459', 'id.458', 'id.456', 'id.450', 'id. 451
age <- c (19, 45, 67, 53, 17, 30, 27, 35)
smoker <- c (TRUE, FALSE, TRUE, TRUE, FALSE, TRUE, TRUE, TRUE)
sex <- c ('f', 'f', 'h', 'h', 'f', 'h', 'f', 'f')
my.db <- data.frame (Id = id, Age = age, Smoker = smoker, Sex = sex); my.db
```

```
##        Id Age Smoker Sex
## 1  id.453  19   TRUE   f
## 2  id.452  45  FALSE   f
## 3  id.455  67   TRUE   h
## 4  id.459  53   TRUE   h
## 5  id.458  17  FALSE   f
```

```
## 6  id.456  30   TRUE   h
## 7  id.450  27   TRUE   f
## 8 id. 451  35   TRUE   f
```

```
dim(my.db); nrow(my.db); ncol(my.db)
```

```
## [1] 8 4

## [1] 8

## [1] 4
```

```
names(my.db)
```

```
## [1] "Id"     "Age"    "Smoker" "Sex"
```

A data frame being a list, we can extract a column using the $ operator preceded by the name of the data frame and followed by the name of the column (or variable), or use the operator [...]

```
my.db$Sex # a column of characters is automatically transformed into a factor
```

```
## [1] f f h h f h f f
## Levels: f h
```

```
my.db[, 2]
```

```
## [1] 19 45 67 53 17 30 27 35
```

```
my.db$Age[my.db$Smoker == FALSE] # simple example of selection
```

```
## [1] 45 17
```

The columns are directly accessible in the workspace (without having to type the name of the data frame and the $) after having *attached* the data frame:

```
attach(my.db)
Age
```

```
## [1] 19 45 67 53 17 30 27 35
```

To display only the first six lines:

```
head(my.db)
```

```
##       Id Age Smoker Sex
## 1 id.453  19   TRUE   f
## 2 id.452  45  FALSE   f
## 3 id.455  67   TRUE   h
## 4 id.459  53   TRUE   h
## 5 id.458  17  FALSE   f
## 6 id.456  30   TRUE   h
```

Similarly, `tail()` creates a data frame with the last six columns.

5.3 Importing and exporting data

Importing data is a fundamental step in data analysis. To load the data stored in a file (texte, .csv, Excel, ...) into the workspace (ie into memory), you can use the basic function `read.table()`. The three most important arguments are:

- `file`: name (and path) of the file, in quotes
- `header`: are the elements of the first row the names of the columns?
- `sep`: character separating the columns

`read.table()` returns a data frame:

```
url1 <- 'https://raw.githubusercontent.com/vittorioperduca/Introduction-to-R/master/da
d1 <- read.table(url1,
                 # the first line contains the name of the variables
                 header = TRUE,
                 # values are separated by ;
                 sep = ';')

head (d1)
```

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

```
url2 <- 'https://raw.githubusercontent.com/vittorioperduca/Introduction-to-R/master/data/heart.tx
d2 <- read.table(url2,
                 header = TRUE,
                 # variables are separated by a tabulation
                 sep = '\t')
dim(d2); names(d2)
```

```
## [1] 270  13
```

```
##  [1] "age"          "sexe"         "type_douleur" "pression"     "cholester"
##  [6] "sucre"        "electro"      "taux_max"     "angine"       "depression"
## [11] "pic"          "vaisseau"     "coeur"
```

For data stored in the `.Rda` or `.Rdata` format, the import is done with `load()` with the argument `file = filename`. For instance download the `Iris.Rda` file at github.com/vittorioperduca/Introduction-to-R/blob/master/data/Iris.Rda to your working directory and then try the following:

```
iris_path <- 'data/Iris.Rda' # replace with the file path
load(iris_path)
```

If you want to load `.Rda` or `.Rdata` files directly from an url, don't forget to use the `url()` function (this was not necessary in `read.table()`).

Data can be exported either to a text file (or .csv, Excel ...) using `write.file()` or to `.rda` and `.Rdata` files at using `save()`. In both cases, the two main arguments are

- `x` = data to save
- `file` = the name of the file (in quotes).

If the dataset is stored (or must be saved up) locally, it is necessary to know (and be able to modify) the working directory:

```
# getwd() # try on your machine!
# setwd('~/Documents') # to move to the Documents directory
```

Remember that in Linux and macOS machines, `~/` is a shortcut for `/Users/username`. For Windows machines, the address syntax is slightly different. For example we use `\` instead of `/`.

5.4 Exercise

Download the text file raw.githubusercontent.com/vittorioperduca/Introduction-to-R/master/data/hepatitis.txt to your working directory.

1. Import the dataset in `R`. Warning: missing data were coded with a `?`, read the documentation of `read.table ()`.
2. Find the number of observations, display the names of the variables and the first six observations. Check that the value of `STEROID` for the fourth observation is missing using the appropriate function.
3. Calculate the mean value of `ALBUMIN` in women and men.
4. Create a variable `NSYMP` counting the number of times a variable is equal to`yes` between `FATIGUE` and`MALAISE`. Pay attention to the format of these two variables!

Chapter 6

Descriptive statistics

6.1 Data used

Basic functions for graphics and descriptive statistics will be illustrated using the `mtcars` dataset available in `R`. For a description of this dataset, see the help: `?mtcars`. We can also get an idea of the class and content of `mtcars` with `str()`:

```
str(mtcars)
```

```
## 'data.frame': 32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

The variable `cyl` represents the number of cylinders with three possible modalities (4, 6 or 8), so it is natural to convert it into a factor:

```
mtcars$cyl.factor <- as.factor (mtcars $ cyl)
attach (mtcars)
```

6.2 Univariate analysis

6.2.1 Quantitative variables

For a quantitative variable, the basic statistics that can be calculated are the minimum, the maximum, the mean, the variance and the standard deviation, the median and the other quantiles (remember that the quantile of order p is the value q such that p is the proportion of observed values less than q).

Try the following commands

```
min(mpg)
max(mpg)
range(mpg)
mean(mpg)
var(mpg)
sd(mpg)
median(mpg)
quantile(mpg)
quantile(mpg, probs = 0.99) # for the 99th percentile
quantile(mpg, probs = c(0.01, 0.1, 0.9, 0.99))
```

If the input vector contains missing data, we can exclude them from the statistic calculation using the option `na.rm = TRUE`.

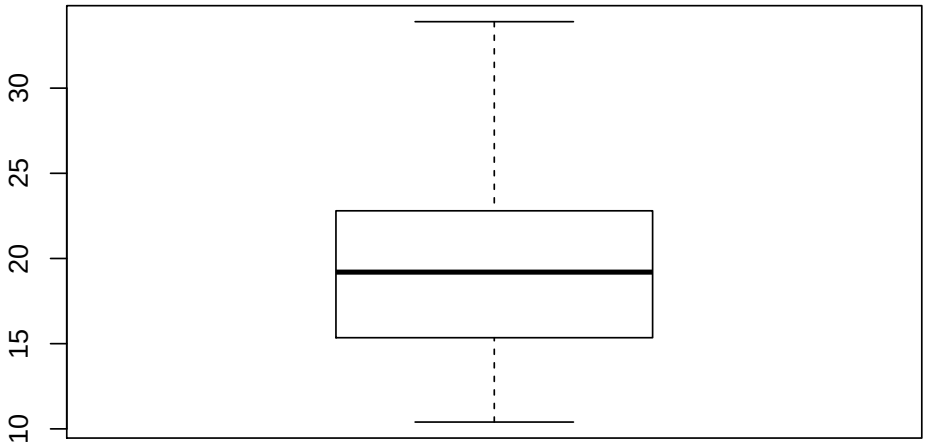
The generic `summary()` function allows to quickly describe the distribution of a sample:

```
summary(mpg)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.40   15.43   19.20   20.09   22.80   33.90
```

An useful tool to show a sample distribution is the *boxplot*:

```
boxplot(mpg)
```

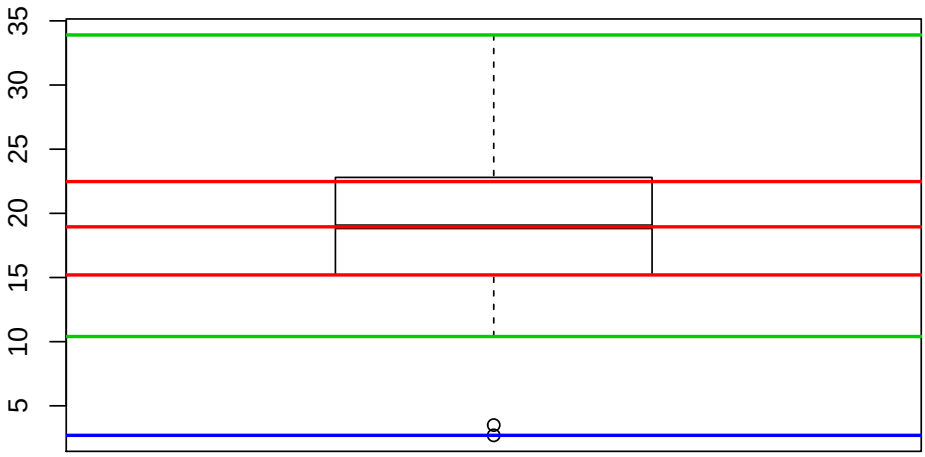


We recall that in a boxplot:

- The height (or length if we choose the option `horizontal = TRUE`) of the box is defined by the first and third quartiles q_1 and q_3.
- The horizontal segment corresponds to the median $m = q_2$.
- The whisker at the bottom starts from q_1 and goes down to the minimum of the sample if there are no extreme points on the left, ie values less than $q_1 - 1.5 \times (q_3 - q_1)$. If there are extreme points on the left, the whisker stops precisely at the smallest point greater than the threshold $q_1 - 1.5 \times (q_3 - q_1)$. In this case, we place the extreme points below the whisker.
- Similarly, the whisker at the top starts from q_3 and goes up to the maximum of the sample if there are no extreme points on the right, ie values greater than $q_3 + 1.5 \times (q_3 - q_1)$. If there are extreme points on the right, the whisker stops at the greatest point less than the threshold $q_3 + 1.5 \times (q_3 - q_1)$, and we place the extreme points above of it.

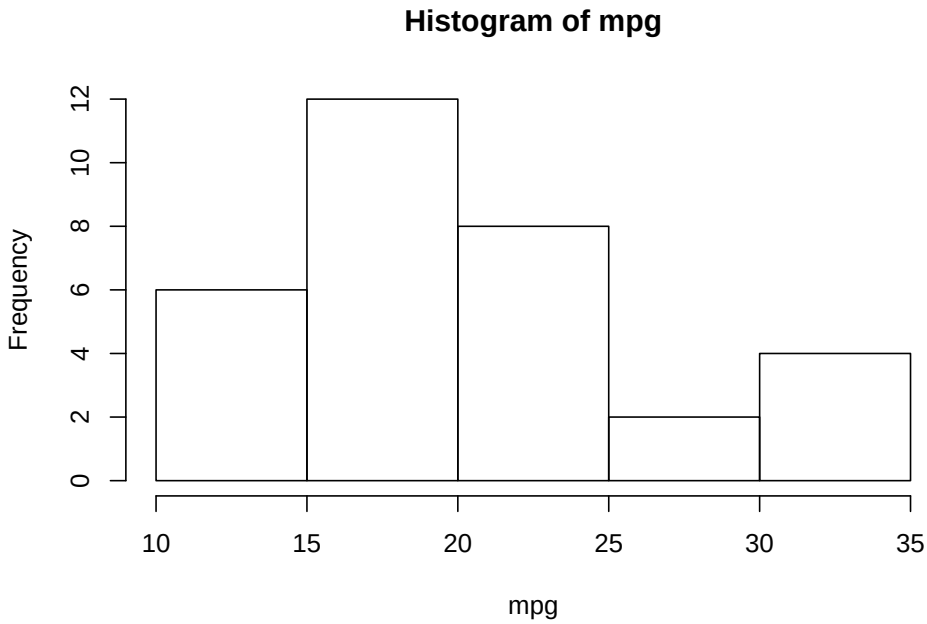
Try the following:

```
u <- c(mpg, 3.5, 2.7) # adding two very low values
boxplot(u)
q1 <- quantile(u, .25); q3 <- quantile(u, .75)
abline(h = median(u), col = 2, lwd = 2)
abline(h = q1, col = 2, lwd = 2); abline(h = q3, col = 2, lwd = 2)
abline(h = max(u), col = 3, lwd = 2)
# u[u < q1-1.5*(q3-q1)] # extreme points on the left
abline(h = min(u[u >= q1-1.5*(q3-q1)]), col = 3, lwd = 2)
abline(h = min(u), col = 4, lwd = 2)
```



The other graphical tool to represent the distribution of a quantitative variable is the histogram:

```
hist(mpg)
```

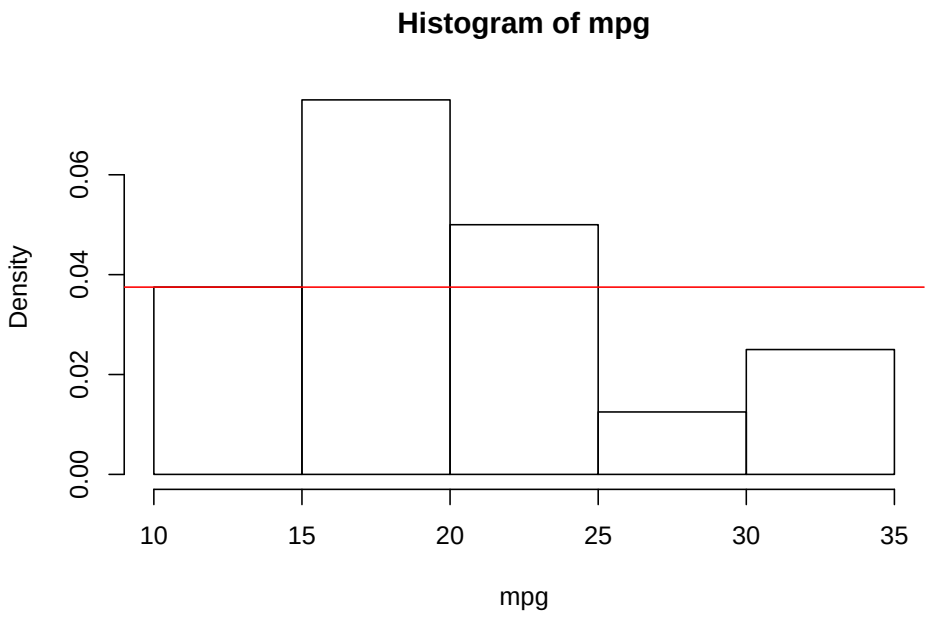



```
# Try the following:
# hist (mtcars$mpg, breaks = 20) # if we want 20 + 1 intervals
# we specify by hand the points which define the intervals:
# hist (mtcars$mpg, breaks = c (10, 15, 18, 20, 22, 25, 35))
```

By default the vertical axis gives the number of observations in each interval.

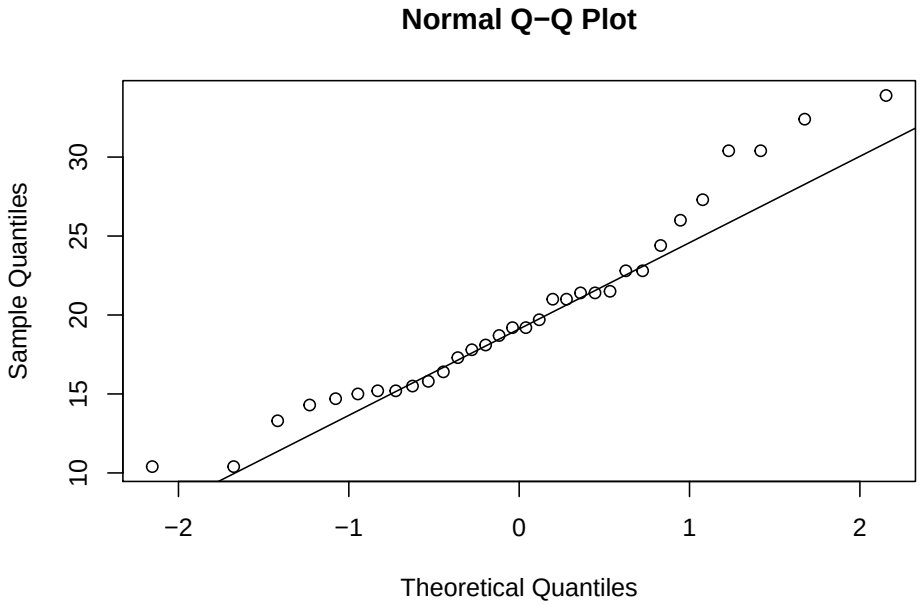
If we want to plot the histogram in the *density* scale, we have to use the option `freq = FALSE`. In this case we recall that the proportion of observations in each interval is given by the area of the corresponding rectangle:

```
hist(mpg, freq = FALSE)
# relative frequency of observations in the [10,15] range:
# the area of the first rectangle is
# mean(10 <= mpg & mpg <= 15)
# to verify this, we divide the area by the base length and show
# that this is precisely the height of the rectangle:
abline(h = mean(10 <= mpg & mpg <= 15)/5, col = 2)
```



We can also compare the distribution of the sample with the theoretical distribution of the standard normal distribution through a *QQ-plot*. This plot compare the empircal quantiles of the sample with the theoretical quantiles of the standard normal, if the observations in the sample were normally distributed than the points in the QQ-plot should be aligned along the bisector:

```
qqnorm(mpg)
qqline(mpg) # the line sample quantiles = theoretical quantiles
```



Remark. Compare the previous QQ-plot to that of the *standardized* sample:

```
mpg.stand <- (mpg-mean(mpg))/sd(mpg) # we standardize mpg
qqnorm(mpg.stand); qqline(mpg.stand)
```

What happened? Note that the distribution of quantiles did not change:

```
hist(quantile(mpg))
hist(quantile(mpg.stand))
```

6.2.2 Qualitative variables

The `summary()` and `table()` functions applied to a factor (or a vector of characters) count the occurrences of the different levels.

```
summary(cyl.factor)
```

```
##  4  6  8
## 11  7 14
```

```
table(cyl.factor)
```

```
## cyl.factor
##  4  6  8
## 11  7 14
```

To get the proportions:

```
prop.table(table (cyl.factor))
```

```
## cyl.factor
##       4       6       8
## 0.34375 0.21875 0.43750
```

If there are missing values, `summary ()` also counts the `NA`:

```
x <- as.factor(c('a', 'a', 'b', 'c', NA, 'c'))
summary(x)
```

```
##    a    b    c NA's
##    2    1    2    1
```

```
table(x)
```

```
## x
## a b c
## 2 1 2
```

```
prop.table(table(x)) # counts are divided by the total number of non-missing values
```

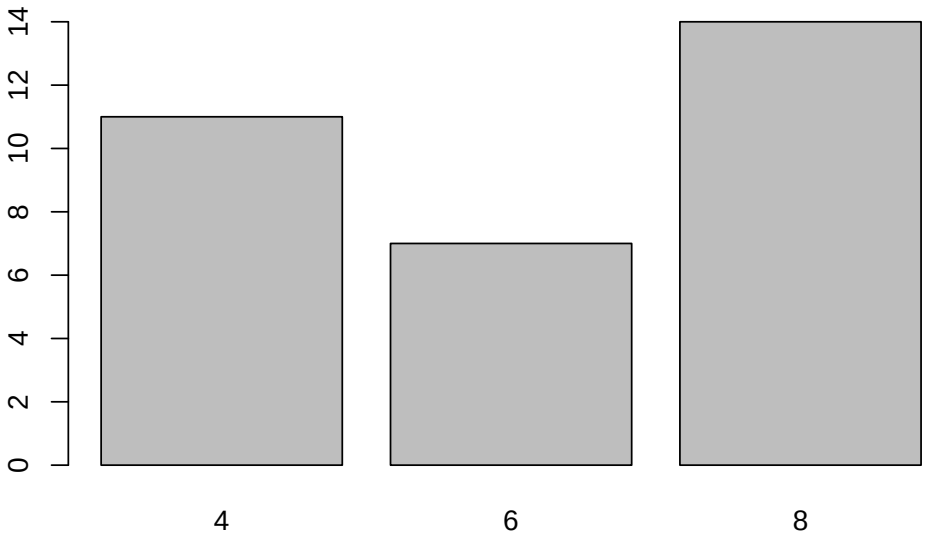
```
## x
##   a   b   c
## 0.4 0.2 0.4
```

```
prop.table(summary(x)) # counts are divided by the total number of values
```

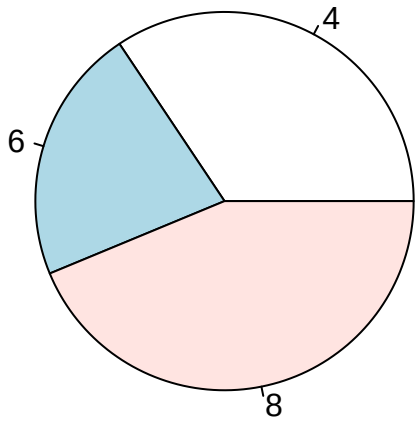
```
##         a         b         c      NA's
## 0.3333333 0.1666667 0.3333333 0.1666667
```

We can use a *bar plot* or a pie chart (not recommended) to represent occurrences:

```
barplot(table(cyl.factor))
```




```
pie(summary(cyl.factor))
```



6.3 Bivariate analysis

6.3.1 Quantitative variable vs quantitative variable

We consider two quantitative variables measured on the same individuals.

For two quantitative variables, we can compute their covariance and linear correlation:

- the covariance measures the average joint deviation from the means of the two variables

- Pearson's linear correlation coefficient ρ measures the intensity of linear dependence. ρ is comprised between -1 and 1: its value is close to 1 (resp. -1) if there is a positive (resp. negative) linear dependence while its value is close to 0 if there is no linear dependence.

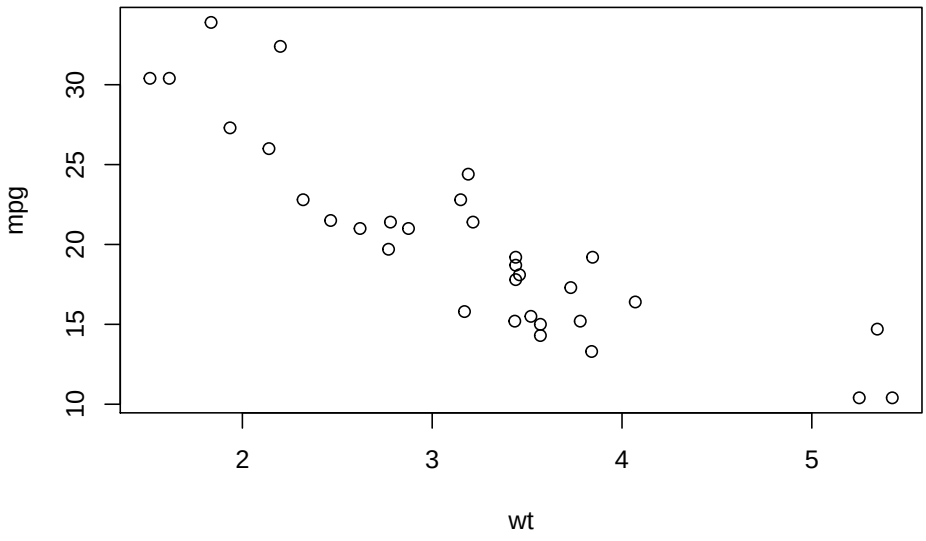
```
cov(mpg, wt)
cor(mpg, wt)
# Try this:
# cov(mpg, wt)/sd(mpg)/sd(wt)
```

Another type of correlation is Spearman's correlation measuring the linear correlation between the *ranks* of the values of the two variables:

```
cor(mpg, wt, method = 'spearman')
# Try this
# cor(rank(mpg), rank(wt))
```

To graphically represent two quantitative variables we will use a a *scatter plot:*

```
plot(x = wt, y = mpg) # x, y = coordinates of the points
```



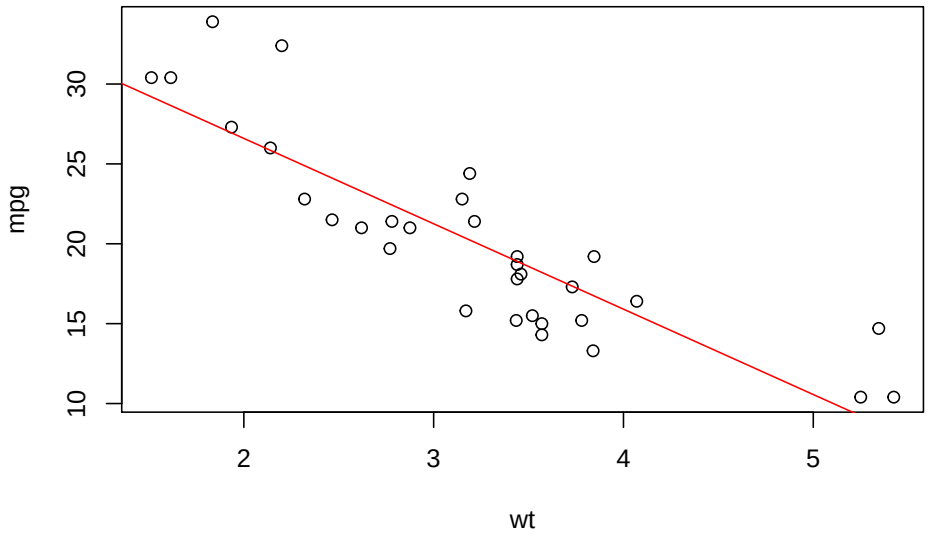

```
# Alternative:
# plot (wt ~ mpg) # where the ~ reads "wt according to mpg"
```

To find the least squares line (ie the linear model that best fits the data) and add it to the data:

```r
m <- lm(mpg ~ wt)
summary(m)
```

```
##
## Call:
## lm(formula = mpg ~ wt)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.5432 -2.3647 -0.1252  1.4096  6.8727
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  37.2851     1.8776  19.858  < 2e-16 ***
## wt           -5.3445     0.5591  -9.559 1.29e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.046 on 30 degrees of freedom
## Multiple R-squared:  0.7528,Adjusted R-squared:  0.7446
## F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-10
```

```r
# try:
# slope <- sum ((wt-mean(wt)) * (mpg-mean(mpg))) / sum((wt-mean (wt))^2)
# intercept <- mean (mpg) - slope * mean (wt)
plot(mpg ~ wt)
abline(m, col = "red")
```



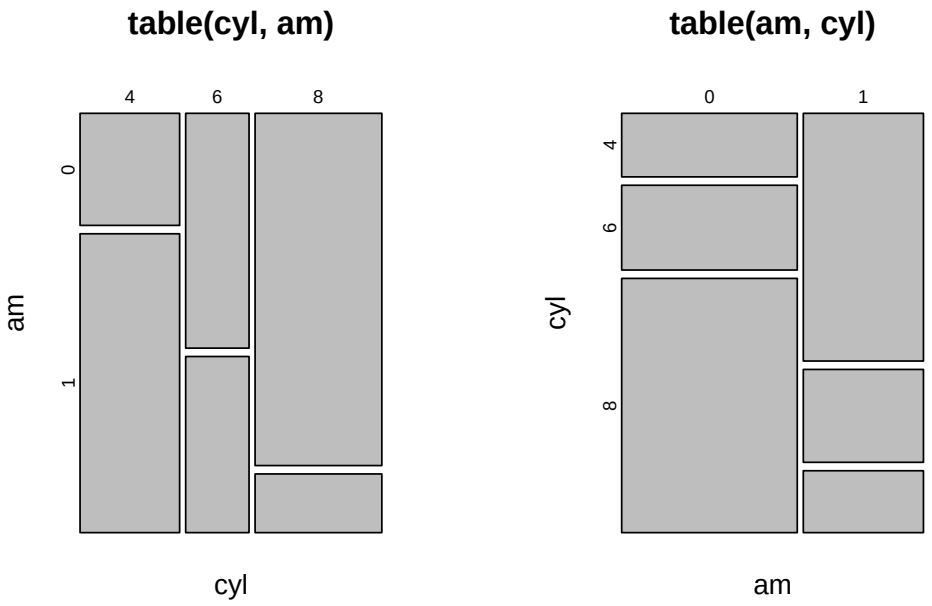
6.3.2 3.2 Qualitative variable vs qualitative variable

We consider two qualitative variables measured on the same individuals. For these variables, we can make contingency tables and calculate the total and marginal relative frequencies (in rows and columns). Try the following:

```
t <- table(cyl, am)
# cyl and am are digital vectors,
# but to build the contingency table
# table() counts their terms.
t
prop.table(t) # Prob(cyl, am)
prop.table(t, margin = 1) # P(am | cyl = 4), P(am | cyl = 6), P(am | cyl = 8)
prop.table(t, margin = 2) # P(cyl | am = 0), P(cyl | am = 1)
```

The corresponding graphic is the *mosaic plot*:

```
par(mfrow = c (1,2)) # to put two charts side by side
mosaicplot(table(cyl, am)) # P(am | cyl = 4), P(am | cyl = 6), P(am | cyl = 8)
mosaicplot(table(am, cyl)) # P(cyl | am = 0), P(cyl | am = 1)
```



6.3.3 Quantitative variable vs qualitative variable

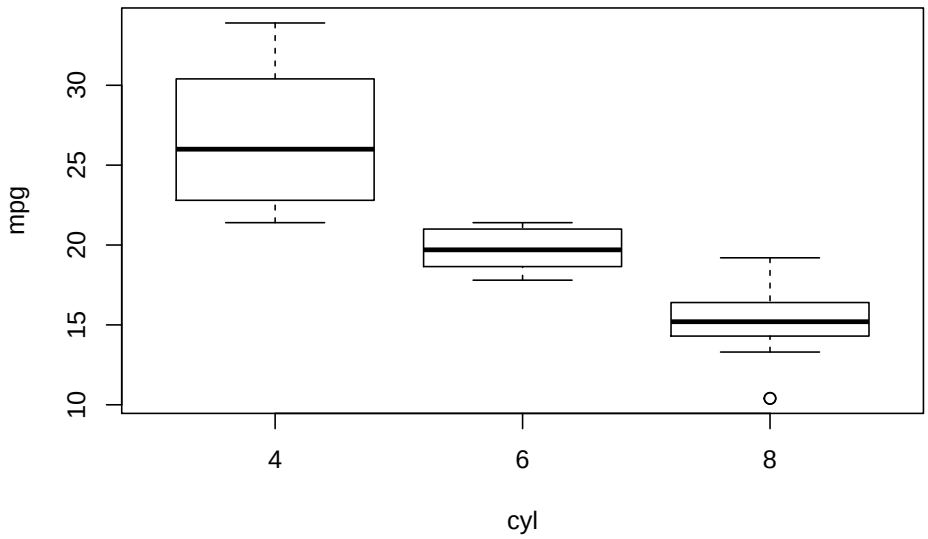
To evaluate a function `f` on the values of a `y` for each level of a factor `x` we use`tapply (X = y, INDEX = x, FUN = f)`. This is very useful when we want

to calculate the statistics of a quantitative variable `y` grouped according to the levels of a qualitative variable`x`. To see this, try the following:

```
tapply(mtcars$mpg, mtcars$cyl, mean) # the average miles per gallon according to the n
tapply(mtcars$mpg, mtcars$cyl, summary)
```

Graphically, we can build the boxplot of the quantitative variable for each level of qualitative variable:

```
boxplot(mpg ~ cyl, data = mtcars)
```



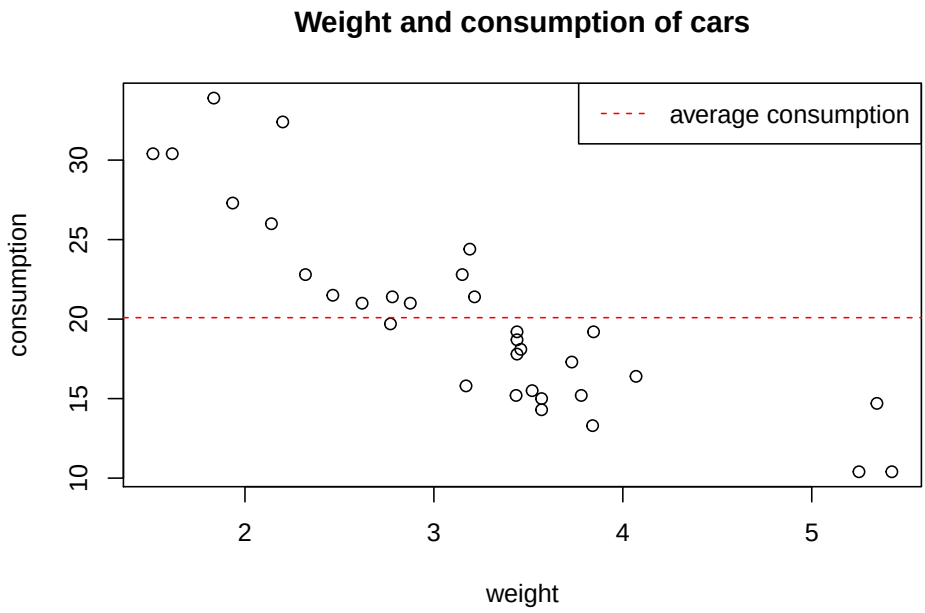
Chapter 7

Base graphics

7.1 Basic constructions and scatter plots

We have already seen how to add a line to a scatter plot, to complete the graph we can specify a title, change the axis labels, and add a legend:

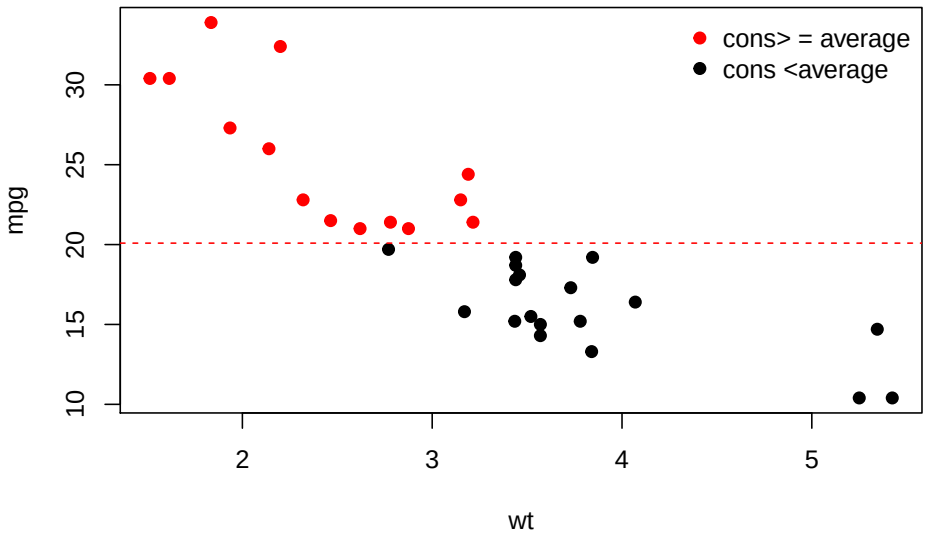
```
plot(wt, mpg,
     main = 'Weight and consumption of cars',
     xlab = 'weight', ylab = 'consumption') # xlab = label x axis
abline(h = mean (mpg), col = 2, lty = 2) # h = horizontal line, col = color, lty = line type
legend(x = 'topright', lty = 2, col = 2, legend = 'average consumption')
```




```
# x = legend location
```

We can also add a third dimension by coloring the points according to a condition:

```
plot(wt, mpg,
     pch = 19, # pch = type of point
     col = (mpg >= mean (mpg)) + 1 # 1 = black, 2 = red
)
abline(h = mean(mpg), col = 2, lty = 2) # h = horizontal line, col = color, lty = line
legend(x = 'topright',
       pch = 19,
       col = 2: 1,
       legend = c ('cons> = average', 'cons <average'), # legend = vector with labels
       bty = 'n' # no frame around the legend
)
```

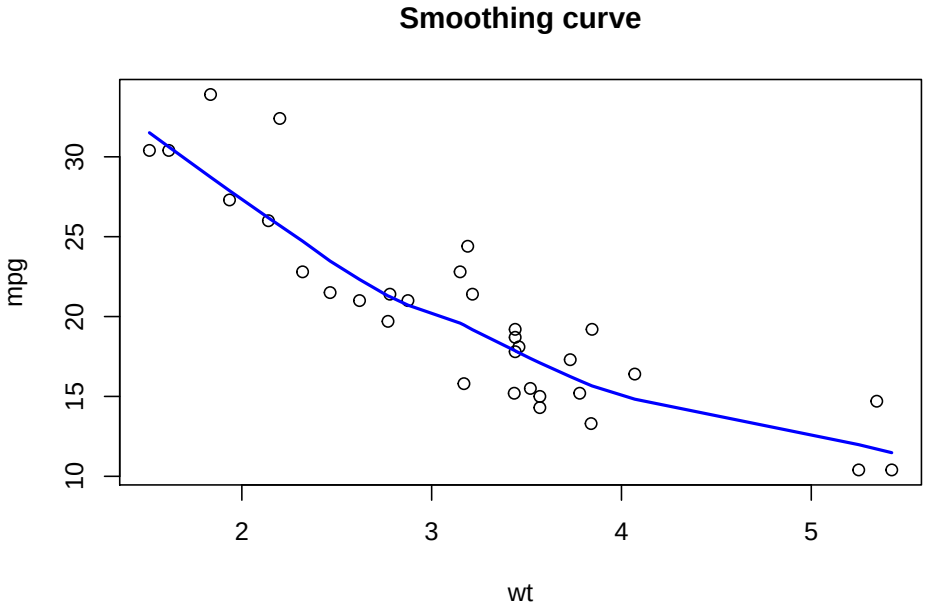


In general, `abline(a, b)` allows to add a line with intercept `a` and slope`b` to an existing plot. For example, an alternative way to add the regression line is as follows:

```
intercept = coef(m)[1]
slope = coef(m)[2]
plot(wt, mpg, main = 'Line of least squares')
abline(a = intercept, b = slope)
```

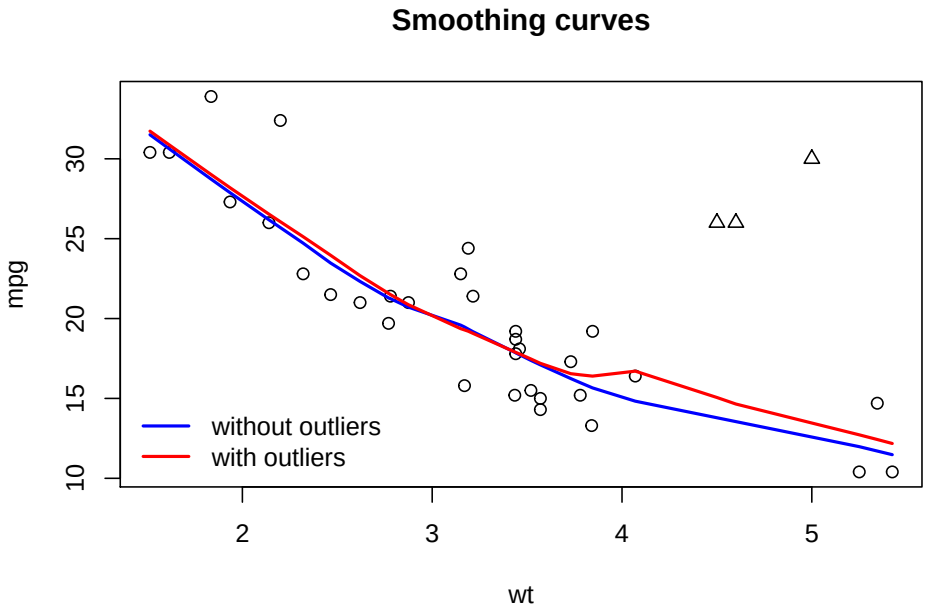
To add any line (not necessarily a straight line) we use the `lines()` function. For example, to add the *lowess* (locally-weighted polynomial regression) smoothing curve:

```
plot(wt, mpg, main = 'Smoothing curve')
lines(lowess (mpg ~ wt), # lines (): to add a curve
      col = 'blue', # same as col = 4
      lwd = 2) # lwd = line thickness
```



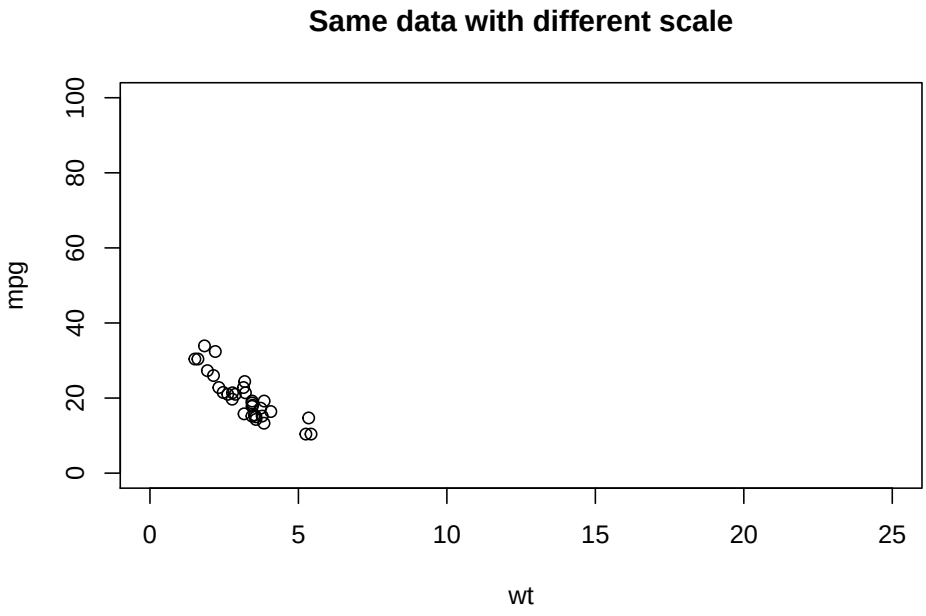
We can add new points using `points()`:

```
plot(wt, mpg, main = 'Smoothing curves')
lines(lowess (mpg ~ wt), col = 'blue', lwd = 2)
outliers = cbind(c(4.5,4.6,5), c(26,26,30))
points(outliers, pch = 2) # pch = 2 for triangles
lines(lowess (c (wt, outliers[, 1]),
              c (mpg, outliers[, 2])),
      col = 2, lwd = 2)
legend (x = 'bottomleft', col = c (4,2), lwd = 2, legend = c ('without outliers', 'with
```



We can control the range of axes using `xlim()` and `ylim()`:

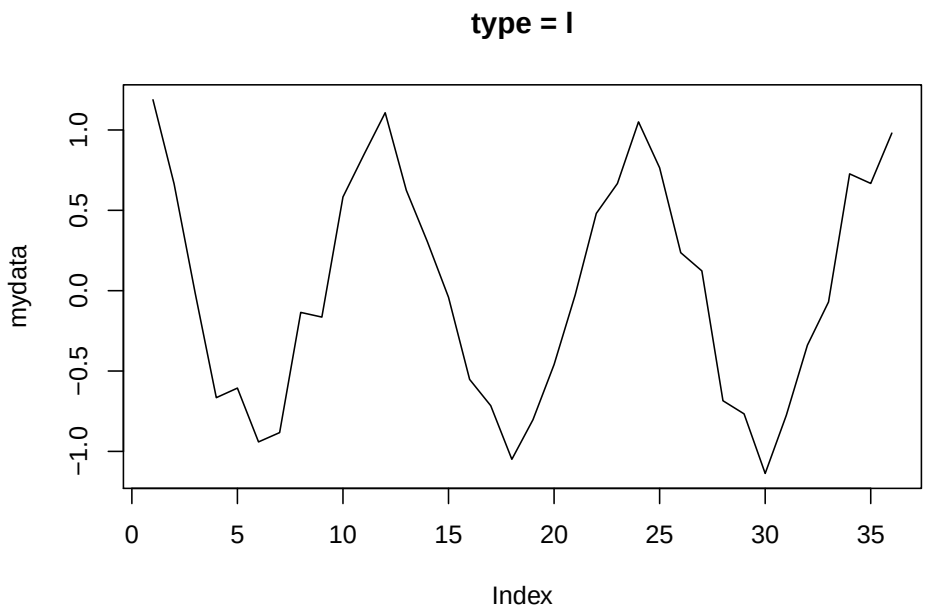
```
plot(wt, mpg, main = 'Same data with different scale',
     xlim = c (0, 25),
     ylim = c (0, 100))
```



7.2 Line graphs

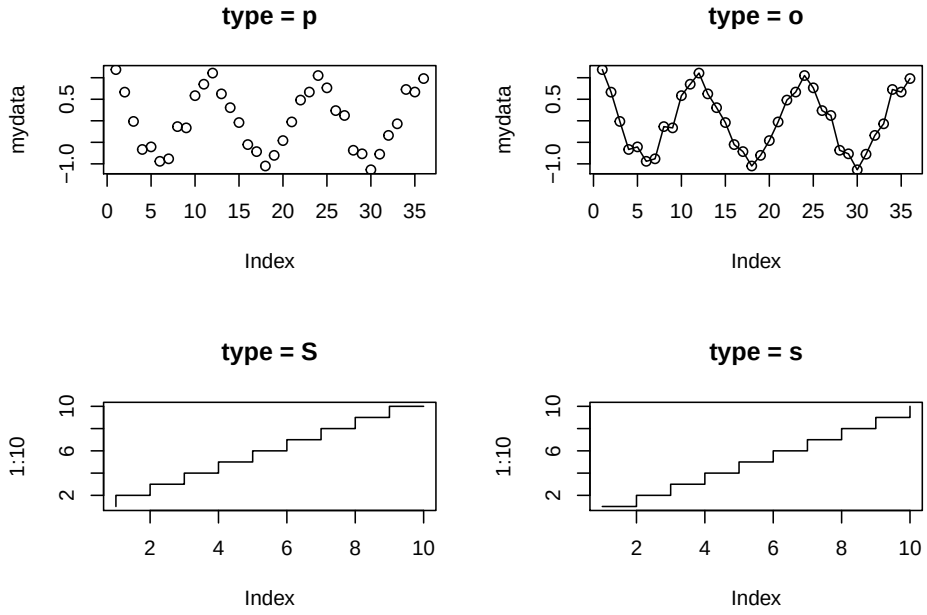
The `lines()` function adds a line to an existing graph and cannot be used to create a new graph. To draw a line connecting the successive elements of a vector we will rather use `plot(x = vector, type = l)`:

```
mydata = cos(2 * pi / 12 * (1:36)) + rnorm (36,0, .2) # data simulation
plot(mydata, type = 'l', main = 'type = l')
```



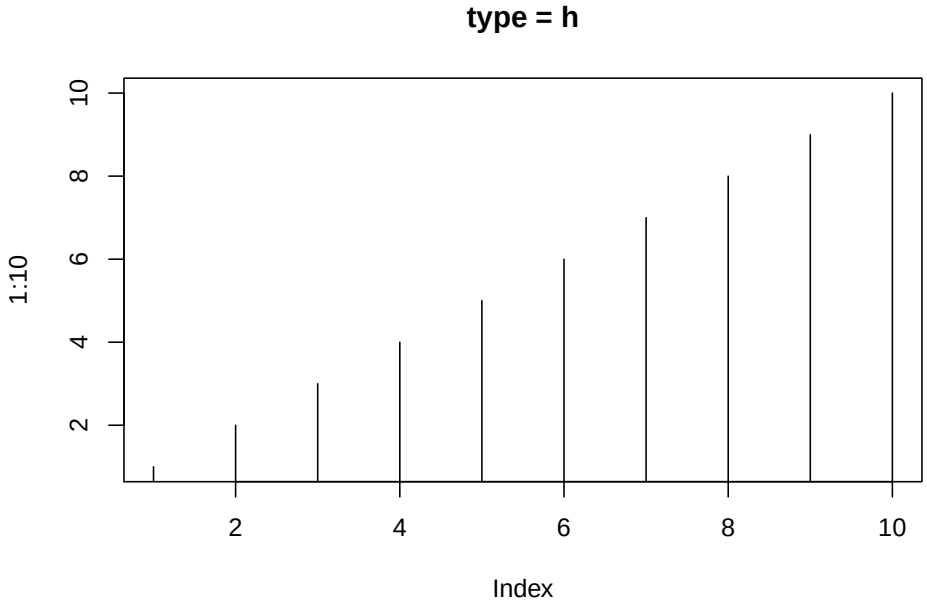
The `type` parameter allows you to connect (or not) the points in different ways:

```
par(mfrow = c (2,2)) # explained in the next section
plot(mydata, type = 'p', main = 'type = p') # default
plot(mydata, type = 'o', main = 'type = o')
plot(1: 10, type = 'S', main = 'type = S') # steps 1
plot(1: 10, type = 's', main = 'type = s') # steps 2
```



With `plot = 'h'` we get a bar plot:

```
plot (1: 10, type = 'h', main = 'type = h')
```



7.3 Graphical parameters

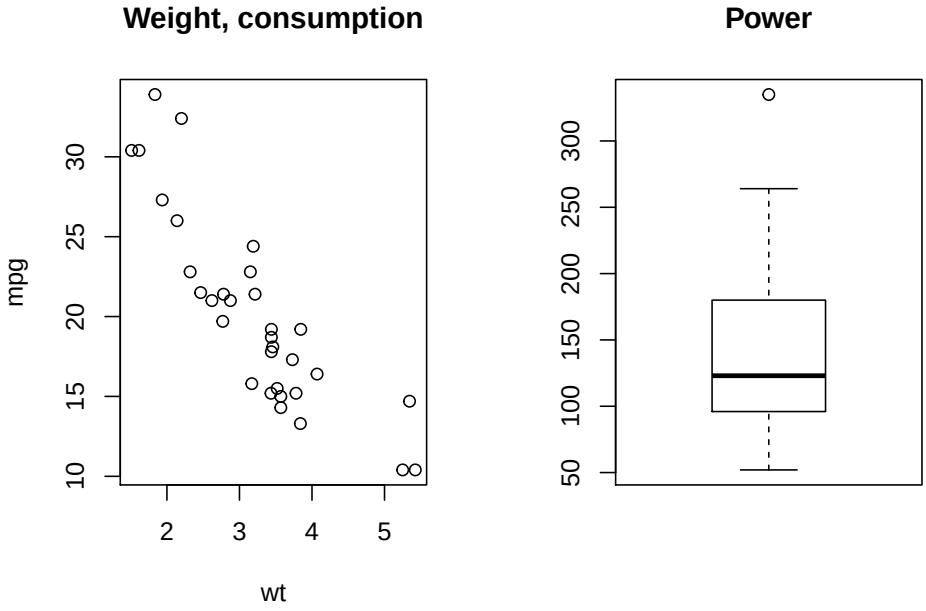
We have already seen that it is possible to control certain graphical elements using parameters:

Element	Parameter
point	`pch`
type of line graph	`type`
color	`col`
line	`lty`,`lwd`
axis label	`xlab`,`ylab`
axis dimensions	`xlim`,`ylim`
label dimensions	`cex`
orientation axes labels	`las`

See `?par` for a description of the values that these parameters can take.

To impose parameters on all the graphics produced during a session, we will use the `par()` function. `par()` is often used to view two or more plots in the same window with the parameter `mfrow = c(l, c)`. In this case the graphs are displayed in a grid with `l` rows and `c` columns. Try the following:

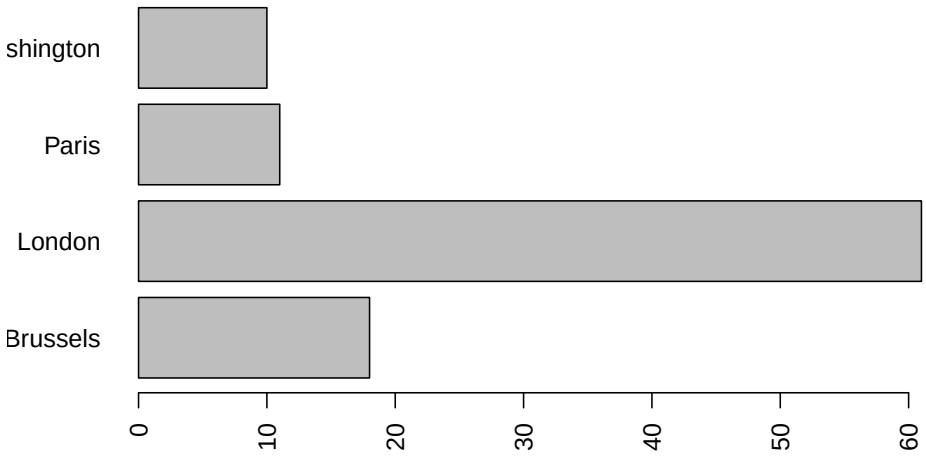
```
# opar = par () # to be able to reset with the initial parameters
par(mfrow = c (1,2))
plot(wt, mpg, main = 'Weight, consumption')
boxplot(hp, main = 'Power')
```



Changes are implemented until when the session is closed, or the graphics engine is reset by `dev.off()` or by clicking on the `Clear all plots` brush in `RStudio`.

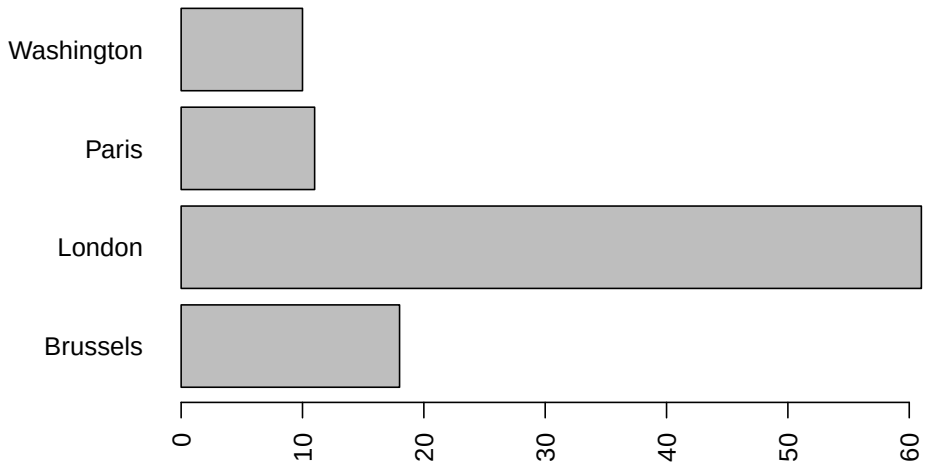
`par()` is also used to change the size of the margins. This is sometimes useful when the labels on the axes do not fit in the window, as in this example:

```
mydata = as.factor(sample (x = c ('Paris', 'London', 'Brussels', 'Washington'),
                           replace = T,
                           prob = c (.1, .5, .2, .1),
                           size = 100)) # simulation of a qualitative variable
barplot(summary (mydata),
        width = .1,
        horiz = T,
        las = 2)
```



We modify the parameter which sets the left margin:

```
par(mar = c(5, 6, 4, 2) + 0.1) # the second component of mar gives the left margin
# Default is c (5, 4, 4, 2) + 0.1
barplot(summary (mydata),
        width = .1,
        horiz = T,
        las = 2)
```



7.4 Histograms

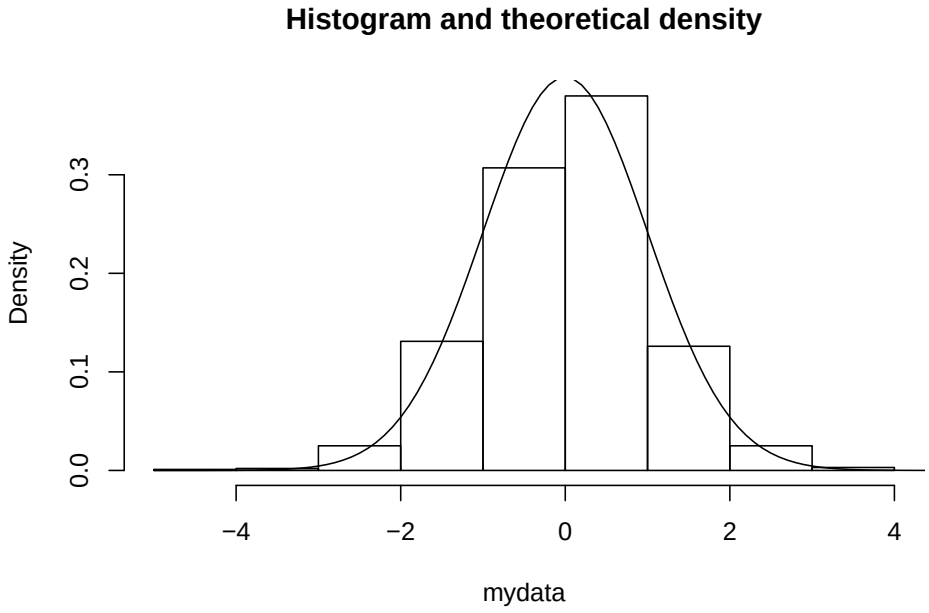
As we have already noted in the subsection 6.2.1, by default, `hist()` displays the histogram with the counts. To display the histogram in the density scale we use the option `freq = FALSE`. In this case, the area of each rectangle will be

equal to the proportion of observations in the corresponding class (so that the total area of all the rectangles is one).

```
par(mfrow = c (1,2))
hist(mpg) # freq = TRUE
hist(mpg, freq = FALSE)
```

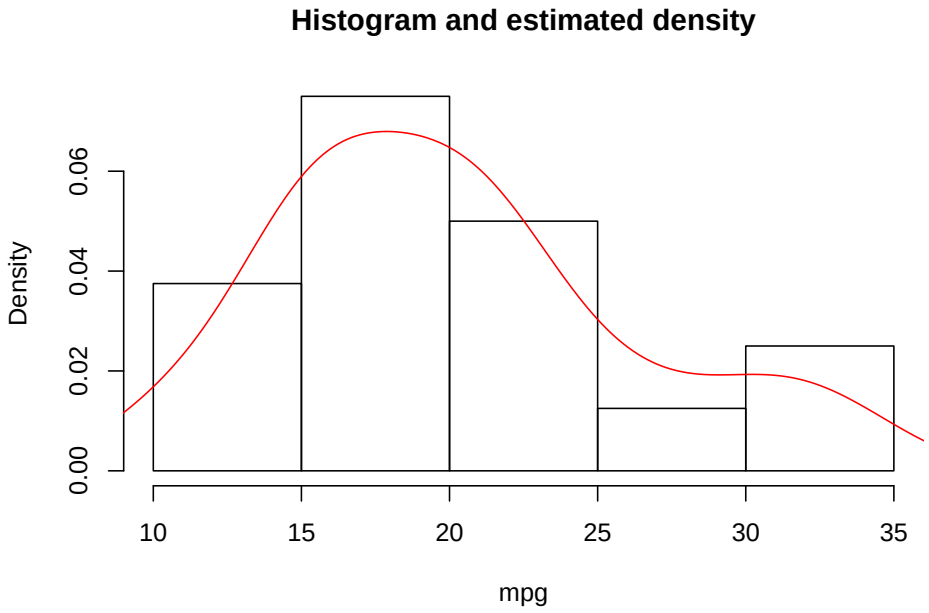
To superimpose the curve of a given density:

```
mydata = rnorm (1000,0,1) # simulation of 1000 observations N (0,1)
hist(mydata, freq = F, main = 'Histogram and theoretical density')
curve(dnorm, # the density function of N (0,1)
      from = -5, to = 5, # the range
      add = TRUE) # because we want to add the density of N (0,1) to the histogram
```



Rather than viewing the histogram of the data, we can show the estimated density (using kernel density estimation methods):

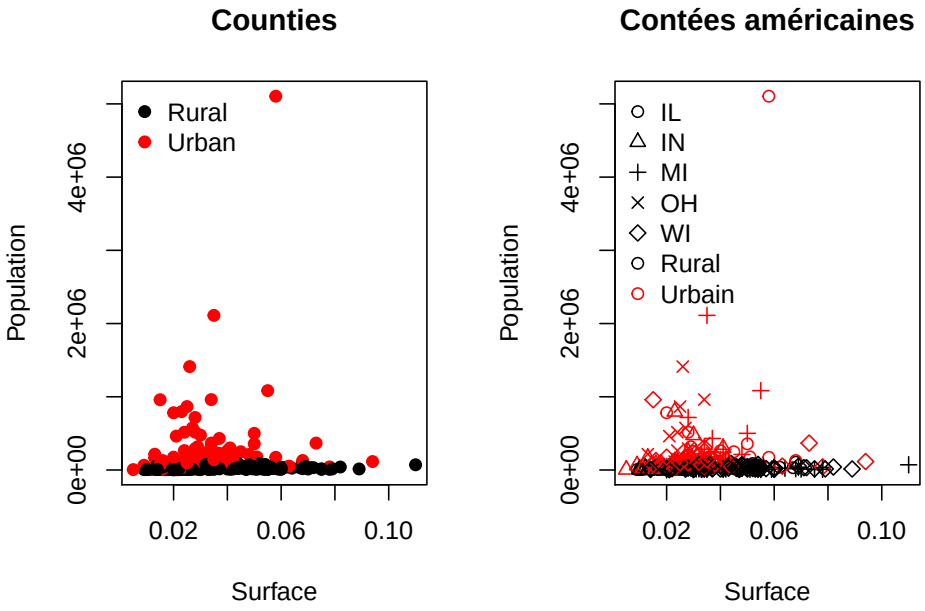
```
hist(mpg, freq = F, main = 'Histogram and estimated density')
lines(density (mpg), col = 2)
```



7.5 Exercises

1. We consider the `midwest` dataset from the `ggplot2` package (an advanced graphics package).

   - Install and charge the package.
   - Consult the help for the description of the variables in `midwest`.
   - Reproduce the following graphics using basic functions:



2. We consider the `adult` dataset available on the site archive.ics.uci.edu/ml/datasets/Adult. The dataset consists of 48,842 rows and 14 columns.

   - Import the data into `R` from the file `adult.data`. Look in the `adult.names` file for the names of the variables.
   - Describe each variable appropriately according to its type.
   - Describe the relationship between the variables `age` and`class`.

Chapter 8

Probability distributions

Let X be a random variable whose distribution is `law`:

- `dlaw(x, parameters)` computes the probability $P(X = x)$ if X is discrete, and the density value $f(x)$ if X is continuous with density f.
- `plaw(q, parameters)` computes the value of the Cumualative Distribution Function (CDF) in q, ie $F(q) = P(X \leq q)$.
- `qlaw(p, parameters)` gives the quantile of order p, that is the value q such that $p = P(X \leq q)$.
- `rlaw(m, parameters)` generates m independent random numbers according to the distribution of X, ie a sample of size m of X.

Consult the help with `?dlaw`.

The `set.seed()` function allows you to set the *seed* of the random number generator. This is useful when you want to obtain identical simulations:

```
runif(1) # uniform draw in [0,1]
```

```
## [1] 0.8366401
```

```
runif(1)
```

```
## [1] 0.7943818
```

```
set.seed(42); runif (1)
```

```
## [1] 0.914806
```

```
set.seed(42); runif (1)
```

```
## [1] 0.914806
```

8.1 Discrete distributions of discrete random variables

Important discrete random variables are

- Binomial variable $\mathcal{B}(n, p)$ counting the number of successes out of n independent trials each with probability p of success: `rbinom(m, size = n, prob = p)`. This law has $n + 1$ distinct possible values: $0, 1, \ldots, n$.
- Bernoulli variable $\mathcal{B}(p)$ reporting the outcome of a binary trial with probability p of success: `rbinom(m, size = 1, prob = p)`
- Geometric variable $\mathcal{G}(p)$ counting the number of failures before success in independent trials: `rgeom(n, prob = p)`
- Poisson variable $\mathcal{P}(\lambda)$: `rpois(m, lambda = lambda)`

An important random process which is very useful to simulate in many situations, is the withdrawl of n balls of different colors out of a box. This can be done with

```
sample(x = box, size = n, replace = TRUE / FALSE, prob = probability
of the different colors)
```

For instance:

```
x <- c('red', 'black', 'green')
sample(x = x, size = 20, replace = TRUE, prob = c(2/10,4/10,4/10))
```

```
##  [1] "red"   "black" "red"   "green" "green" "green" "black" "green" "green"
## [10] "green" "green" "red"   "black" "green" "red"   "red"   "black" "green"
## [19] "green" "red"
```

8.2 Continuous distributions

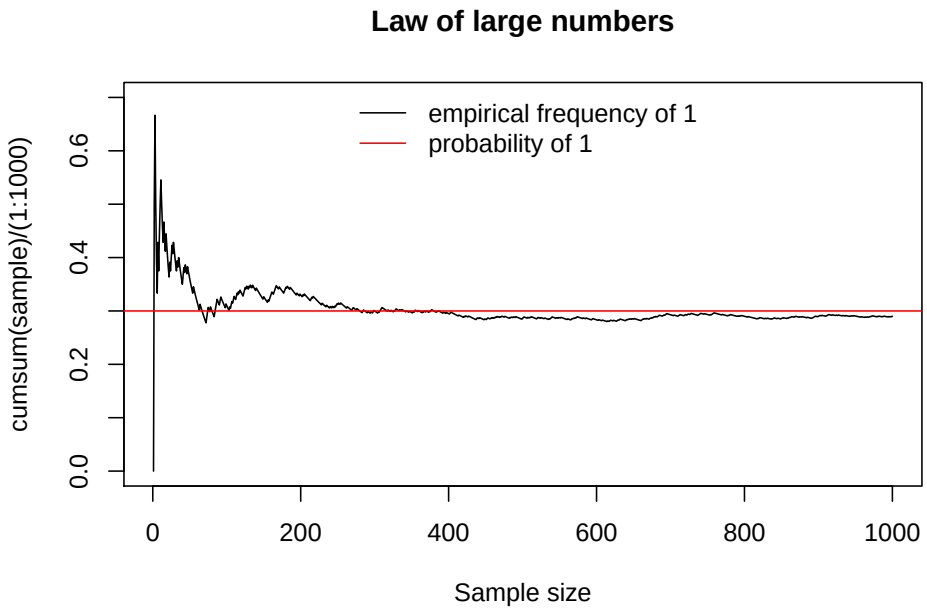
Important continuous distributions are:

- Uniform distribution $\mathcal{U}_{[a,b]}$ in the interval $[a, b]$: `runif(m, min = a, max = b)`
- Normal variable $\mathcal{N}(\mu, \sigma^2)$: `rnorm(m, mean = mu, sd = sigma)`
- Exponential law $\mathcal{E}(\lambda)$: `rexp(m, rate = lambda)`
- Chi-square law with r degrees of freedom $\chi^2(r)$: `rchisq (m, df = r)`
- Student's law with r degrees of freedom: `rt(m, df = r)`

8.3 Example: the Law of Large Numbers

We toss n times a biased coin such that the probability of getting head is $p = 0.3$ and then we calculate the proportion of heads obtained. The Law of Large Numbers says that this proportion gets closer and closer to p as n gets larger.

```
# 1: head, 0: tail
sample = sample(x = c(0,1), size = 1000, replace = T, prob = c(.7, .3))
plot(cumsum(sample)/(1:1000),
     type = 'l',
     main = 'Law of large numbers',
     ylim = c (0,0.7),
     xlab = 'Sample size')
abline (h = .3, col = 2)
legend (x = 'top',
       lty = 1, col = 1: 2,
       legend = c ('empirical frequency of 1', 'probability of 1'),
       bty = 'n')
```



Notes:

- To get the number of heads as a function of n we used the `cumsum()` function. Try the following:

```r
# for each n we count how many 1s there are in the first n positions:
cumsum(c(0,0,1,0,1,1,0))
# for each n we calculate the proportions of 1s in the first n positions
cumsum(c(0,0,1,0,1,1,0))/1:7
```

- Instead of `sample()` we could have used `rbinom(n = 1000, size = 1, prob = .3)`

Bibliography

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC, Boca Raton, Florida. ISBN 978-1138700109.

Xie, Y. (2020). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.20.