Pasteur course

"Introduction to data analysis"

Part

"Introduction to R"
## CONTENTS

1. Introduction ........................................................................................................................................... 3

2. How to get RStudio and documentations ................................................................................................. 3
   2.1. RStudio installation for Windows, Mac OS and Linux ......................................................................... 3
   2.2. Helpful online documentations ........................................................................................................ 3

3. Rapid presentation of RStudio ....................................................................................................................... 3
   3.1. Console (bottom left) ........................................................................................................................ 4
   3.2. Editor pane (top left) ....................................................................................................................... 4
   3.3. Environment and History pane (top right) ....................................................................................... 5
   3.4. Tools pane (bottom right) ............................................................................................................... 5
      3.4.1. Files tab ...................................................................................................................................... 5
      3.4.2. Plots tab .................................................................................................................................... 5
      3.4.3. Packages tab ............................................................................................................................ 5
      3.4.4. Help tab .................................................................................................................................. 7
      3.4.5. Viewer tab .................................................................................................................................. 7

4. Description of the main features of R ........................................................................................................ 7
   4.1. The instruction ................................................................................................................................... 7
   4.2. The objects ...................................................................................................................................... 8
      4.2.1. Data objects ............................................................................................................................ 8
      4.2.2. Functions ................................................................................................................................ 9
   4.3. Operators ...................................................................................................................................... 10
   4.4. Special features ............................................................................................................................ 11

5. First step: R is a calculator ......................................................................................................................... 12

6. Manipulation of data objects .................................................................................................................... 12
   6.1. Creation of data objects .................................................................................................................. 12
      6.1.1 Creation by writing ................................................................................................................... 13
      6.1.2. Creation by importing a data text file ................................................................................... 18
   6.2. Description of data objects ............................................................................................................ 22
      6.2.1. The `length()` function ....................................................................................................... 22
      6.2.2. The `mode()` function .......................................................................................................... 22
      6.2.3. The `class()` function .......................................................................................................... 24
      6.2.4. The `summary()` function ................................................................................................... 24
   6.3. Analysis and modification of data in a data objects ........................................................................... 25
   6.4. Import instructions .......................................................................................................................... 35
   6.5. Export of data objects ...................................................................................................................... 36

7. Manipulation of conditional or repeated instructions .............................................................................. 36
   7.1. Conditional features of R ............................................................................................................... 36
   7.2. Looping features of R ...................................................................................................................... 38

8. R is a graphic editor ................................................................................................................................... 39
   8.1. Discovery of the graphics window .................................................................................................... 39
   8.2. Different kinds of graphics .............................................................................................................. 40
      8.2.1. Description of the most used graphic functions ..................................................................... 40
      8.2.2. Common arguments of these functions .................................................................................. 40
   8.3. Adding items to a graph ................................................................................................................... 42
   8.4. Graphical parameters ..................................................................................................................... 43
   8.5. Colors ............................................................................................................................................ 45
   8.6. Graphic export .................................................................................................................................. 47
INTRODUCTION TO R

1. Introduction

R was created by two members of the Auckland university's Department of Statistics: Ross Ihaka and Robert Gentleman, also known as R & R (Ihaka and Gentleman, 1996). It has since evolved and today, its development is carried out by a core of a dozen statisticians (core team) and by additional contributors from around the world.

The R software uses its own language, derived from the computer language S, specially developed for Statistics. R can be designated as R software, or R language or R environment. Its success stems from the fact that (1) it is free, (2) it is compatible with the most used operating systems (Windows, Mac OS, Linux), (3) documentation is available online, and (4) it is very powerful. Most statistical calculation, functions and graphics are available in the basic version. If it does not, you can find additional features into packages developed by statisticians and available on the web. Lastly, for those who require particularly sophisticated tools, or functions that do not exist, you can create your own functions to achieve what you want.

The R software (and associated packages) can be downloaded at http://www.r-project.org/. During the class, we will rather use another software, called RStudio. This software is a so-called integrated development environment (IDE), derived from the classical R software. As shown below, it offers a friendly interface that should satisfy most R users.

2. How to get RStudio and documentations

2.1. RStudio installation for Windows, Mac OS and Linux

The RStudio software can be downloaded at https://www.rstudio.com site. The supplemental files needed can be found here: \atlas.pasteur.fr\@hubbioit/Enseignement/CoursPhD

2.2. Helpful online documentations

In the main page of the classical R software (http://www.r-project.org/), on the left menu, click on: (1) Documentation Manuals, giving you access to the official documentation of R, or (2) Documentation Other. Here, you will find the documentation of contributors in various languages, more particularly in Contributed documentation. Documentation specifically dedicated to RStudio can be found here: https://support.rstudio.com/hc/en-us. The RStudio team also provides useful documents, like https://www.rstudio.com/wp-content/uploads/2016/10/r-cheat-sheet-3.pdf.

3. Rapid presentation of RStudio

Compared to the classic R console, the RStudio window is made of four subwindows, called panes, facilitating the use of R. The current paragraph provides a quick description of these panes. Some of the terms used here will be addressed in details in the subsequent paragraphs of the document.
3.1. Console (bottomleft)

The main pane is the console, located in the bottomleft part of the RStudio window. It corresponds to the console found in the classical R software with some additional features (e.g., when directly tapping the code into this pane). It is considered as the main pane because the three others are not essential for R computation and running. A "->" symbol, called the prompt, is present below the introduction text. To the right of the prompt, you will find the cursor. Instructions can be directly typed, or copy-pasted, into this pane, but as explained below, we will take the benefit from the editor pane to execute instructions, rather than to directly use the console. Lastly, a STOP icon appears in the top of the console pane when running code. Click on it to kill a run if necessary.

3.2. Editor pane (topleft)

The topleft pane is an editor. For beginners, it is recommended to write your code in this editor, rather than directly in the console, then execute your lines of code from here. If errors are signaled during code running, correct them in the text editor and rerun the code. This procedure has three advantages: (1) significant time savings because you avoid re-writing the lines that have already been executed, (2) if you have a lot of instructions, they will be easier to read, manipulate and change in a text editor (for example, you can easily find a command line with a keyword in the text editor, using Ctrl + f in windows or Command + f in Mac OS), (3) There is no warning in R except when leaving the software. This last point is important to know. You can change or lose data without realizing it. Thus, saving the code written in an editor is a good programming practice that has to be applied also when using RStudio.

Note that conventional shortcuts of Windows (Ctrl + c, ctrl + v, etc.) and Mac OS (Command + C, Command + v, etc.) can be used in RStudio. This means that code written in an external
file can be copy-pasted into the editor pane for modification, or copy-pasted into the console pane for direct execution.

It is also possible to write an instruction that will execute a script contained in an external document, either directly from the console, or from the editor pane, provided that the document is a "text file" (meaning, ending with the extension ".txt" or ".R" for Windows and Mac OS). This procedure, named code sourcing, is useful and will be addressed in paragraph 6.5.

3.3. Environment and History pane (topright)

This topright pane displays either the history of all the code executed in the console (History tab) or information about all the objects created (Environment tab) in the working environment, also called **Workspace**.

3.4. Tools pane (bottomright)

Five tabs are present in the bottomright pane: respectively the Files, Plots, Packages, Help and Viewer tab.

3.4.1. Files tab

The Files tab allows to browse in the computer files. Note that clicking on the arrow, close to the name Console in the Console pane, shows the directory used by RStudio to store the data saved (called **the Home directory**). This working directory can be changed using the bar menu Session/Set Working Directory/Choose Directory. It is very important to identify the location of the Home directory, as it is possible to save all the code present in the History pane (save icon in the topright History pane described above). This will generate a .Rhistory file in the Home directory. It is also possible to save all the objects created, either via the bar menu Session/Save Workspace As, or when quitting RStudio. This will generate a .RData file in the Home directory. Both the .Rhistory and the .RData files are automatically imported when opening RStudio.

3.4.2. Plots tab

The Plots tab displays the history of all the plots drawn. The Arrow buttons allow the navigation among the different plots. The Export button saves the displayed plot.

3.4.3. Packages tab

The Packages tab shows all the packages available in your computer (located in the library directory of R). A package is a collection of tools that is not systematically included in the basic installation of RStudio. To get a package from the web, click on the install icon, then specify the package name desired in the new window:
The package will be installed in the default directory (C:/Program Files/R/R-3.2.2/library for Windows and /Library/Frameworks/R.framework/Versions/3.2/Resources/library for Mac OS) and will appear in the Packages tab list.

If this method does not work (more particularly when you are not administrator of the computer), an alternative is to recover by hands the different packages, but this can be fastidious because some packages require the downloading of additional packages (see the example below). Go to the R website at https://www.r-project.org/. In the home page, click on Download CRAN (left side of the page). Select the mirror site https://cran.univ-paris1.fr/ (beware: the next described pages can differ depending on the mirror sites).

For Windows, click Download R for Windows, then on contrib. Choose the version of R used in RStudio, written in the console when RStudio is started, (for instance 3.2/). Then, select the desired package, in its most recent version (for instance, coin_1.0-24.zip). A "zip" file is saved on your hard disk. Unzip it. Find the unzipped folder without the version number (for example, corner) and drag it into the library folder described above, or put it on the desktop if you are not administrator of your computer.

For Mac OS, click Download R (Mac) OS X, then contrib (at the bottom of the page). Choose the version of R used in RStudio, written in the console when RStudio is started, (for instance 3.2/). Then, select the desired package, in its most recent version (for instance, coin_1.0-24.tgz). A "tgz" file is saved on your hard disk. Unzip it. Find the unzipped folder without the version number (for example, corner) and drag it into the library folder described above, or put it on the desktop if you are not administrator of your computer.

For Linux, click Download R for Linux, then on the operating system used (for instance ubuntu/), and follow the instructions described, which depend on the operating system.

To be active in RStudio, a package must be "loaded" into the working environment. For that, click on the desired packages in the Package tab of RStudio, on coin for instance:

```r
> library("coin", lib.loc="C:/Program Files/R/R-3.2.2/library")
```
Le chargement a nécessité le package : survival
Le chargement a nécessité le package : splines

As mentioned above, the package coin requires additional packages, which must be downloaded from the web if not present in the library folder of R, or on the desktop.

If located on the desktop, the package can be "loaded" this way:
```r
> library(coin, lib.loc = "C:/Users/Gael/Desktop/"
```  
indicates the absolute pathway of the coin folder, which is addressed in paragraph 6.1.2.

3.4.4. Help tab

The Help tab displays information about the functions already available in R (see paragraph 4.2.2.).

3.4.5. Viewer tab

The Viewer tab can be used to view local web content.

4. Description of the main features of R

Before introducing how to use R to create and manipulate data sets, let us see the different features of the R language. The most important is to distinguish between the data objects, that contain the data to be analyzed, from the other features of R. These other features, including functions, operators and special features, are used to create, manipulate and analyze data objects. The separation between functions, operators, and special features, fundamental for advanced programmers, can be neglected for people starting with R.

4.1. Instructions

Any succession of characters, numbers and symbols typed in the console is an instruction for R. We also call this "command line", "line of code", or "expression".

The "->" symbol is called the prompt. This is the invitation to write instructions. To its right is the cursor. If "->" is replaced by "++", it means that the instruction typed is not complete. Missing elements must be added. In Windows, press "Escape" or the STOP icon to exit the "++" and return to the prompt "->". For Mac, press "Escape".

Once a sequence of characters is typed in the console or in the editor window, R treats it as an instruction. There is one exception: everything written between the "#" symbol and the end of the line (return sign produced by the "Enter" key of the keyboard) is not recognized by R. Very convenient to provide explanations when the code is long, this feature will be widely exploited below.

In the console, a written instruction is executed when the "Enter" key is pressed. In the editor window, it is executed when the written instruction is selected and the "run" icon clicked. If the written instruction is not complete, the prompt "->" is replaced by "++" which means that R is waiting for the end of the instruction. Note that two punctuation marks are reserved for the succession of instructions: the semicolon ";" and the comma ",". The first is used to separate two complete instructions (instead of using the "enter" key), and the second is used to
separate statements within functions (we will see that below). Caution: R uses the Anglo-Saxon system of number writing. Decimals are separated by a dot and not by a comma.

In this document, instructions are easy to identify: each line of code is in font "courier bold style" and starts with the prompt "->". When the line of code is followed by an explanation, written behind the "#" symbol, the font is "courier italic". Finally, the results are in font "courier normal style" and immediately follow the code that generated them. For reasons of space, semicolons ";" are often used in the code that will be presented later. But you can easily replace them with a newline ("Enter" key) when you type this code. Finally, with a few exceptions which will be indicated, you can add (or not) spaces and line breaks between elements of code.

Seven characters, widely used in R, are hard to find on a French Mac AZERTY keyboard: ~, [, ], {, }, | and \. They are obtained by the respective commands "alt n", "alt shift (", "alt shift )", "alt (", "alt )", "alt shift L" and "alt shift /".

4.2. The objects

R is an object oriented language, meaning that it works with objects. An object is like a box that can be filled with anything. You just have to remember two types of important objects: data objects and functions.

4.2.1. Data objects

When one wishes to perform a statistical test, the first thing to do is to create or import the table containing the data into R. These data will be placed in an object that becomes a data object. This object is temporary and will exist as long as you do not quit R. To be definitively saved (meaning placed in the .Rdata file), you must save the workspace or save when leaving R. Three things can happen to this data object (Figure 1). It can be:
(1) Described, which allows to check the size, number and nature of the items.
(2) Modified, to be suitable for subsequent analysis.
(3) Used, more particularly in a statistical test.
Figure 1: diagram showing how data objects are processed in R. The object is represented by a box. The arrows indicate the operations that can be performed on a data object: creation / import, or description, or modification, or use. The example shows three values of maize plant height (in centimeters) created or imported into R. The object is then described by requesting the number of elements it contains, or used to get the average of the values, or modified by removing a value and changing the name. The first two actions generate an output.

The data objects must have a name. In R, the name should only be made of classic Anglo-Saxon letters, numbers and dot. If you do not apply this rule, you increase the chances of generating bugs. The name must start with a letter. Do not use spaces or special characters including letters with accents, cedilla, punctuation other than dot, etc. Several examples:

Masse.grains
grandmere24
etoile.slach.pointdexclamation

And not:
Masse/grains
Masse,grains
grandmère24
*./.!
etoile.slach.pointd'exclamation
etoile.slach.point dexclamation

Beware: R is case sensitive. Thus Masse.grains and masse.grains will be considered as two different objects.

4.2.2. Functions

The other important objects to know are functions. Two aspects make functions easy to distinguish from a data object. First, the function name is followed by brackets. Example: `sum()`. Second, the box of a function does not contain data but lines of code. To view the content of a function, simply type the function name without brackets. Example:

```r
> sum
function (... , na.rm = FALSE) .Primitive("sum")
```

The functions are objects that perform a procedure: modification of an object, statistical computation, graphical display, etc. The presence of the brackets allows the execution of functions. Most of the time, the parameters of a function, called arguments must be specified within the brackets. Let us compute, for instance, the sum of 1 and 4 with the `sum()` function.

```r
> sum(1,4)
[1] 5
```

Note that, in this example, the "5" result of the sum performed is preceded by "[1]". It's a R result display, which appears for certain functions, that facilitates the counting of output items when they are numerous. The number between brackets indicates the serial number of the item located at the beginning of each line. For the first line, it is always 1.

Beware: in R, commas are used to separate the arguments inside the brackets of functions. Any other use may generate error messages or bugs. The most common error comes from decimal numbers. R uses the Anglo-Saxon system of number writing. Decimal places are separated by a dot and not by a comma.
All the functions that beginners need are already available in R. But you also have the possibility to create your own functions, using the special feature `function(){}` (not addressed here).

Help functions are very valuable, especially the `help()`, function or the equivalent operator `?`, that you will use all the time. It explains, in a new window, how to use a function and how to set the parameters between brackets. It is very practical as the help window consists almost always of the following paragraphs:

- **Description**: indicates the nature of the function
- **Usage**: shows how we must write the function
- **Arguments**: explains the terms used between the brackets of the function
- **Details**: provides more information on how the function operates
- **Value**: details the output data when the function is executed
- **References**: indicates the origin of the function
- **See Also**: shows related functions (very useful)
- **Examples**: provides examples that you can copy and paste into the console

**Example:**
```r
> help(sum) # or ?sum. You can leave the brackets empty or not. Examples: sum() or ?sum(1, 4). With help(), it is mandatory to put only the name of the function: help(sum). The quotation marks are optional: help("sum") or "sum". However, only the name of the function must be used if the quotes are present: "sum()" produces an error message
```

### 4.3. Operators

Operators are particular features of R that can also act on data objects. Different types of operators are available: arithmetic, logical, comparison, extraction and assignment.

**Arithmetic operators** are used for simple calculations. Examples:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division

**Logical operators** are comparison tools (see paragraph 6.3). Examples:

- `&` and
- `|` or (keys "Alt Gr 6" on Windows and "alt shift L" on Mac OS, using an AZERTY keyboard)

**Comparison operators** compare data objects or data object elements (see paragraph 6.3). Examples:

- `<` less than
- `<=` less than or equal to
- `>` greater than
- `>=` greater than or equal to
- `==` equal to (do not confuse with `=`, see below)
- `!=` different

The extraction operators `[,]`, `[[, ]]` and `$` allow to extract some elements of a data object (see paragraph 6.3).
Let us focus on the assignment operator. Indeed, this is the operator that creates objects. Three different operators are available:

<- Assignment of the right to the left ("less than" sign followed by a dash sign, which form an arrow that indicates the direction of the assignment)

-> Assignment from left to right (dash sign followed by a "greater than" sign)

= similar to <-

To create an empty object:
> x<-NULL

To assign the 4 number to the object named x:
> x<-4

Be careful when writing code: it is possible to put spaces between the object and the assignment operator (x<-4 or x <-4), but no carriage return ("Enter" key).

In R, there is no need to create an empty object before filling it. The object does not exist but it is created during the assignment. The first step x<-NULL, is thus not necessary. To verify that x exists, we can use the \texttt{ls()} function, which lists all the objects created in memory:
> ls()
[1] "x"

In the result display, the value between brackets, before "x", is here to facilitate the counting of created objects (see the previous paragraph 4.2.2).

Be careful with the names of objects during assignment: if the object already exists, the assignment will overwrite it by default, without warning.

Finally, let us see the specific operator ~. It means explained by (e.g., \( y \sim x \) means data object \( y \) explained by data object \( x \)). It is used primarily in the linear analysis, when variables have to be explained using a model. But it is also used in some graphics functions such as \texttt{stripchart()} and test functions, such as \texttt{t.test()}.

The list of the most common operators can be obtained using \texttt{?Syntax} or using the help on one of the operators in each category. Note that operators must necessarily be written between double quotes. Example:
> ?"+" # or help("+")

4.4. Special features

In R, some words are reserved for specific use (see Help \texttt{?...} For the full list). There are two categories: those that run nothing and those that can execute instructions. Examples from the first category:

\texttt{NA} Missing value (\textit{Not Available})
\texttt{NaN} result of impossible arithmetic computation, like 0 divided by 0 (\textit{Not a Number})
\texttt{Inf} infinity (\texttt{Inf} for minus infinity)
\texttt{NULL} empty object
\texttt{TRUE} logical argument "True"
\texttt{FALSE} logical argument "False"
Those from the second category will be treated in Paragraph 7. Note that the \texttt{function()} \texttt{()} feature, allowing to create functions, also belongs to this category.

R also possesses a few constants. Example:

\begin{verbatim}
letters the 26 letters of the alphabet in lower case
pi \pi = 3,141593
\end{verbatim}

Warning: the name of a reserved word cannot be used to create an object. However, it is possible for constants, which can generate bugs. For instance, R will take in priority the \texttt{pi} object rather than the constant \texttt{pi} if both exist. The complete list of constants is available with the help of one of them, \texttt{?pi} for instance.

5. First step: R is a calculator

Let us see some very simple examples of calculation.

Division:
\begin{verbatim}
> 10/3
[1] 3.333333
\end{verbatim}

Square root:
\begin{verbatim}
> 25^0.5 \texttt{# or sqrt(25)}
[1] 5
\end{verbatim}

Decreasing series of number:
\begin{verbatim}
> 5:2
[1] 5 4 3 2
\end{verbatim}

Standard calculation using \pi = 3,141593:
\begin{verbatim}
> 8*(4+2^2)/3-pi
[1] 18.19174
\end{verbatim}

More complicated calculation using the formula
\[
\frac{|54 - 93|}{\sqrt{\cos(1 - 0.62^2)}} \times e^{-8/3} \times 6! + \log_{10}(\sqrt{4}) - 2.10^2:
\]
\begin{verbatim}
> abs(54-93)/cos(1-0.62)^0.5 * exp(-8/3) * factorial(6) + log10(4^(1/3))-2e+2
[1] 1824.846
\end{verbatim}

6. Manipulation of data objects

This paragraph briefly addresses how R can be used on data objects. Warning: use the Anglo-Saxon system when writing numbers (separate decimals with a dot and not a comma). In R, the comma is used to separate arguments within functions.

6.1. Creation of data objects

The assignment operators \texttt{<-} and \texttt{=} have already been seen in paragraph 4.3. Let us start with the different types of data objects that we need and the different functions that create them (see paragraph 6.2.2 for an explanation of the mode):
6.1.1 Creation by writing

Vectors

Creation of a simple vector:
> x<-4
> x
[1] 4

Creation of a classical vector (series of values in a single dimension):
> height<-c(130,300,170) # numeric vector. To name the elements: height<-c(a = 130, b = 300, c = 170). The name will appear over the values
> height
[1] 130 300 170

> colour<-c("Red","Yellow","Yellow") # vector of characters
> colour
[1] "Red" "Yellow" "Yellow"

> rooting<-rep(c("Weak", "Intermediate", "Strong"), c(1,3,2)) # the rep() function replicates the elements of the first vector the number of times indicated in the second vector
> rooting
[1] "Weak" "Intermediate" "Intermediate" "Intermediate" "Strong" "Strong"

Creation of an empty vector of non-null size:
> empty.vec <- vectorErreur ! Signet non défini.("character", 5) # character vector. The value 5 indicates 5 empty quotes to generate. For a numeric vector, write "numeric". In this case, the items displayed are zeros. For a logical vector, write "logical". In this case, the items displayed are FALSE. This procedure is useful for generating a vector that can subsequently be gradually filled
> empty.vec
[1] "" "" "" "" ""

Matrices

Matrices (series of two-dimensional values) can be created starting from vectors. Row and column numbers are indicated by numbers in square brackets: the number before the comma specifies the line number, and the number after the comma the column number.

Creation of a matrix:
chiffre<-c(1:15)  # can also be written chiffre <-1: 15
matrix.1<-matrix(chiffre, ncol=3)  # the ncol argument states that the matrix has 3 columns. The matrix is filled by columns from left to right and up to down, by default. Warning: if the number of elements in chiffre divided by the number of columns does not generate an integer, the function matrix() recycles the number of elements in chiffre to complete matrix.1. Test this with chiffre <-c(1:13)

matrix.1  # numeric matrix
[,1]   [,2]   [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15

Another way to create a matrix:
matrix.1<-1:15  # the matrix.1 object is first a vector
dim(matrix.1)<-c(5, 3)  # dim() converts the matrix.1 vector to a matrix with 5 rows and 3 columns (specified by the vector c (5, 3)). If the specified dimensions (here 5 rows x 3 columns = 15 cases) do not match exactly the number of elements in matrix.1, the dim() function does not recycle and an error message is generated

matrix.1
[,1]   [,2]   [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15

Creation of a matrix with three rows and filling by rows:
matrix.2<-matrix((1:15)*100,nrow=3, byrow=TRUE)  # the vector (1:15) * 100 is introduced directly into the matrix() function

matrix.2
[1,]  100  200  300  400  500
[2,]  600  700  800  900 1000
[3,] 1100 1200 1300 1400 1500

Creation of an empty matrix of non-null size:
empty.mat<-matrix(nrow=2, ncol=4)  # this procedure is useful for generating a matrix that can subsequently be progressively filled

empty.mat
[1,] NA NA NA NA
[2,] NA NA NA NA

Arrays

Arrays (series of n-dimensional values) can be created starting from vectors. The row and column numbering is identical to that of the matrices. Additional dimensions levels are indicated above each plane. Example of creation of a table with three rows, five columns and two levels by filling the array with letters in alphabetical order:
array.1<-array(letters, dim=c(3,5,2))  # the letters feature of R gives the alphabet and dim = c (3,5,2) indicates that the array has 3 rows, 5 columns and 2 levels of 3rd dimension. As for matrix(), the array() function recycles the elements of letters to complete the filling of the array
array.1  # array of character mode: the characters are always presented between quotes, it's a way to identify the mode of the object. By only putting the dim = c (3,5,2) argument, it creates an empty array filled with NA
The first level corresponds to the lowest floor of a cube. It is indicated by ", , 1". You can add as many additional dimensions as necessary (use the above code with the dim = c (3,5,2,3) argument for example). The number of commas indicates the size shown: before the first comma, it is the number of rows, after the first comma, it is the number of columns, after the second comma, the number of levels in the third dimension, etc.

**Data frames**

Data frames (series of values in columns) can be created starting from vectors or matrices. The columns are always variables. By default, the name of a column starts with a X or V, followed by the column number. Otherwise, it is the name of the vector that becomes the column. The mode is always the same in a given column, but it can be different between columns. Warning: the characters are not displayed between quotes because a character column is considered as a qualitative variable by default. This means that the column is of type "factor" by default (the I() function prevents this event if necessary). Of note, if you are unable to perform operations on numeric columns of a data frame, it might be because they are of type "factor".

**Creation of a data frame from a matrix:**

```r
> dataframe.1<-data.frame(matrix.2)
> dataframe.1
 X1   X2   X3   X4   X5
1 100  200  300  400  500
2 600  700  800  900 1000
3 1100 1200 1300 1400 1500
```

**Creation of a data frame from a series of numeric and character vectors:**

```r
> height<-c(130,300,170)
> mass<-c(1431,1468,1398)
> nb.grains<-c(320,290,147)
> colour<-c("Red", "Yellow", "Yellow")
> dataframe.2<-data.frame(height, mass, nb.grains, colour) # all vectors must have
>     the same number of elements (here 3). Otherwise the function produces an error
>     message. Replace the area shaded in grey by I(colour). Nothing changes in the
>     data frame displayed but the colour column is no longer a factor, it is a
>     vector of characters. This is confirmed when using the instruction
>     mode(dataframe.2$colour), which displays "numeric" without I() and "character"
>     with I() (see paragraphs 6.2.2 and 6.3 for further explanation). Column names
>     can be directly specified. Example: data.frame (a = height, b = mass, c =
>     nb.grains, d = colour)
> dataframe.2
   height mass nb.grains colour
1     130  1431       320   Red
2     300  1468       290   Yellow
3     170  1398       147   Yellow
```

**Creation of a data frame of non-null size:**

```r
> empty.mat<-matrix(nrow=2, ncol=4) # formation of an empty matrix
```
```r
> empty.dataframe<-as.data.frame(empty.mat) # this procedure is useful for
generating a data frame that can subsequently be progressively filled
> empty.dataframe
  V1  V2  V3  V4
1 NA NA NA NA
2 NA NA NA NA

Lists

Lists (sets of data) join all kinds of objects in different compartments: vectors, matrices, arrays, data frames, lists, factors and tables. In lists, each compartment is specified with a number between double brackets.

Creation of a list from objects created earlier:
> list.1<-list(x, height, matrix.1, dataframe.2) # to name the compartments:
    list.1<-list(a = x, b = height, c = matrix.1, d = dataframe.2)
> list.1
[[1]]
 [1] 4
[[2]]
 [1] 130 300 170
[[3]]
[,1] [,2] [,3]
[1,]   1   6  11
[2,]   2   7  12
[3,]   3   8  13
[4,]   4   9  14
[5,]   5  10  15
[[4]]
 height mass nb.grains colour
1  130  1431   320   Red
2  300  1468   290  Yellow
3  170  1398   147  Yellow

Creation of a list of non-null size:
> empty.list<-vector("list", 1) # the value 1 indicates the number of empty
    compartments to create. This procedure is useful for generating a data frame
    that can subsequently be progressively filled
> empty.list
[[1]]
NULL

Factors

A factor is a kind of vector adapted to qualitative variables. It can be created starting from a character vector.

Creation of a factor:
> rooting<-rep(c("Weak", "Intermediate", "Strong"), c(1,3,2))
> rooting
[1] "Weak" "Intermediate" "Intermediate" "Intermediate" "Strong" "Strong"
> factor.1<-factor(rooting)
> factor.1 # note the difference between the rooting vector above and the factor
    factor.1: no quotes and presence of Levels
Levels: Weak Strong Intermediate
```
The `factor()` function considers the characters as the different categories of a qualitative variable, also referred to as "levels". In the result, "Levels" shows these different categories (in alphabetical order when applied to a vector of characters). Factor objects are very convenient for level counting, i.e. for determining the number of individuals in the different levels of a variable.

Special feature of factor objects: elements of factors are stored as numbers so that each number corresponds to a class of "Levels". It is therefore possible to create a factor with a vector of numbers, by indicating a label to associate to each number, using the `levels` and `labels` arguments:

```r
> rooting.2 <- factor(c(5, 0, 10, 5, 40, 5, 0), levels=c(0, 5, 10, 40),
                      labels=c("Weak", "Intermediate", "Strong", "Very.strong"))
```

The `levels` argument indicates the presence of four numbers, such that 0 = "Low", 5 = "Middle" = 10 "Strong" 40 = "Very.strong". Use the argument `ordered = TRUE` if you want to report `rooting.2` as an ordinal variable.

```r
> rooting.2
Levels: Weak Intermediate Strong Very.strong
```

Note that this time, "Levels" does not display the levels according to the alphabetical order but according to the writing order in the `labels` argument, which is convenient in some cases.

### Tables

A table object is a contingency table. Tables can be created starting from factors but also from vectors or matrices of characters. The rows and columns take the name of each class (factor) or of each character string (vector and matrix). The table dimensions can be greater than two (more than two factors or vectors used, see the arrays).

Creation of a table starting from a vector:

```r
> rooting <- rep(c("Weak", "Intermediate", "Strong"), c(1,3,2))
> table.1 <- table(rooting)
> table.1
# the same is obtained if the vector is transformed into factor or matrix
rooting
Weak   Strong  Intermediate
1      2             3
```

Creation of a table starting from two vectors (which must have the same number of elements):

```r
> parcel <- c("North", "North", "North", "South", "North", "South")
> table.2 <- table(rooting, parcel)
> table.2
# the same is obtained if the two vectors with the same number of
# elements are transformed into two factors or into two matrices
parcel
rooting        North South
Weak            1     0
Strong          1     1
Intermediate    2     1
```

It is easier to understand the counting when the two vectors are placed into a data frame (which, by the way, converts them into factors):

```r
> dataframe.3 <- data.frame(rooting, parcel)
> dataframe.3
```

---

**Note:** The document is copyrighted and for non-commercial use only.
rooting   parcel
1       Weak       North
2       Intermediate       North
3       Intermediate       North
4       Intermediate       South
5       Strong       North
6       Strong       South

> table.2<-table(dataframe.3) # can also be written xtabs(~ rooting + parcel, dataframe.3) or xtabs(~., dataframe.3). Warning: the names of the columns in the data frame are no longer indicated in the table if the data frame is divided into columns. Example: table(dataframe.3[,1], dataframe.3[,2])

> table.2

<table>
<thead>
<tr>
<th>parcel</th>
<th>rooting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Weak</td>
</tr>
<tr>
<td>North</td>
<td>1</td>
</tr>
<tr>
<td>south</td>
<td>0</td>
</tr>
<tr>
<td>strong</td>
<td>1</td>
</tr>
<tr>
<td>intermediate</td>
<td>2</td>
</tr>
</tbody>
</table>

Finally, the table reports the number of identical rows in the frame data.

Creation of an empty table of non-null size:

> empty.mat<-matrix(nrow=2, ncol=4)
> empty.table<-as.table(empty.mat) # this procedure is useful for generating a table that can subsequently be progressively filled

> empty.table

A B C D
A
B

6.1.2. Creation by importing a data text file

One of the advantages of R is the possibility to import existing data files coming from other softwares, under certain conditions. The simplest way, for beginners, is to have "text" formatted files (i.e., "txt" files). Note that most softwares can save a file in the "text" format (Excel for instance). However, these "text" files must have an appropriate structure. First, a good statistical usage is to define your data table using the columns for the variables, and using the rows for each entity on which measurements are performed (Table 1).

Table 1: measured variables of 10 maize plants coming from the same population

<table>
<thead>
<tr>
<th>Maize stalk n°</th>
<th>n¹</th>
<th>n²</th>
<th>n³</th>
<th>n⁴</th>
<th>Random Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>height (cm)</td>
<td>total mass (g)</td>
<td>total number of grains</td>
<td>grain mass (g)</td>
<td>colour of grains</td>
</tr>
<tr>
<td>1</td>
<td>199</td>
<td>1431</td>
<td>320</td>
<td>92.1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>205</td>
<td>1468</td>
<td>290</td>
<td>89.4</td>
<td>red</td>
</tr>
<tr>
<td>3</td>
<td>173</td>
<td>1398</td>
<td>147</td>
<td>42.6</td>
<td>yellow</td>
</tr>
<tr>
<td>4</td>
<td>233</td>
<td>1622</td>
<td>138</td>
<td>43.2</td>
<td>red</td>
</tr>
<tr>
<td>5</td>
<td>206</td>
<td>1428</td>
<td>166</td>
<td>44.1</td>
<td>yellow</td>
</tr>
<tr>
<td>6</td>
<td>261</td>
<td>1574</td>
<td>151</td>
<td>50.7</td>
<td>yellow</td>
</tr>
<tr>
<td>7</td>
<td>155</td>
<td>1215</td>
<td>293</td>
<td>88.2</td>
<td>yellow / red</td>
</tr>
<tr>
<td>8</td>
<td>214</td>
<td>1457</td>
<td>345</td>
<td>108.6</td>
<td>yellow / red</td>
</tr>
<tr>
<td>9</td>
<td>174</td>
<td>1368</td>
<td>234</td>
<td>78.8</td>
<td>red</td>
</tr>
<tr>
<td>10</td>
<td>191</td>
<td>1431</td>
<td>320</td>
<td>92.1</td>
<td>-</td>
</tr>
</tbody>
</table>

Additionally, writing rules have to be respected in the file. The table 1 example is not usable in R: the header is not simple; special characters are present in the column names (accents,
brackets, space, etc.); Finally, in the dataset, slashes (/), accents, spaces and commas could generate bugs.

Here is a way to proceed with a "Excel" spreadsheet on Windows and Mac OS:
1. Use only Anglo-Saxon letters, numbers as well as the dot. Try to other characters, specially spaces or commas (in the Anglo-Saxon system, decimals are separated by a dot and not by a comma). Use the "Find and Replace" search tool of Excel to detect and remove them.
2. The first row is reserved for the variable names.
3. Each following row is an individual.
4. If individuals have a name, write that name in the first column.
5. The missing data should be noted NA (Not Available). This is the reserved word of R corresponding to the missing data.
6. Then save your file in the "text" format (tab delimited) (*.txt).

With a text editor:
1. Same recommendations as above.
2. Separate the elements of each line by one and only one tab.

Table 2 is an example of a table that can be imported into R.

<table>
<thead>
<tr>
<th>Individual</th>
<th>Height</th>
<th>Mass</th>
<th>Nb.grains</th>
<th>Mass.grains</th>
<th>Colour</th>
<th>Ear.germination</th>
<th>Rooting</th>
<th>Tilt</th>
<th>Attack</th>
<th>Parcel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>Weak</td>
<td>NA</td>
</tr>
<tr>
<td>2</td>
<td>199</td>
<td>1431</td>
<td>320</td>
<td>92.1</td>
<td>Red</td>
<td>No</td>
<td>Intermediate</td>
<td>No</td>
<td>No</td>
<td>North</td>
</tr>
<tr>
<td>3</td>
<td>205</td>
<td>1468</td>
<td>290</td>
<td>89.4</td>
<td>Yellow</td>
<td>No</td>
<td>No</td>
<td>Intermediate</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>173</td>
<td>1398</td>
<td>147</td>
<td>42.6</td>
<td>Yellow</td>
<td>No</td>
<td>Weak</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>233</td>
<td>1622</td>
<td>138</td>
<td>43.2</td>
<td>Red</td>
<td>No</td>
<td>Very.strong</td>
<td>Yes</td>
<td>No</td>
<td>North</td>
</tr>
<tr>
<td>6</td>
<td>206</td>
<td>1428</td>
<td>166</td>
<td>44.1</td>
<td>Yellow</td>
<td>No</td>
<td>Intermediate</td>
<td>Yes</td>
<td>No</td>
<td>North</td>
</tr>
<tr>
<td>7</td>
<td>261</td>
<td>1574</td>
<td>151</td>
<td>50.7</td>
<td>Yellow</td>
<td>No</td>
<td>Intermediate</td>
<td>Yes</td>
<td>No</td>
<td>North</td>
</tr>
<tr>
<td>8</td>
<td>155</td>
<td>1215</td>
<td>293</td>
<td>88.2</td>
<td>Yellow.Red</td>
<td>No</td>
<td>Weak</td>
<td>No</td>
<td>No</td>
<td>North</td>
</tr>
<tr>
<td>9</td>
<td>214</td>
<td>1457</td>
<td>345</td>
<td>108.6</td>
<td>Yellow.Red</td>
<td>No</td>
<td>Intermediate</td>
<td>No</td>
<td>No</td>
<td>North</td>
</tr>
</tbody>
</table>

Table 2: Table 1 converted for R import

When you will be used to R, you will be able to use the arguments of the import functions (see read.table() below) and import more complex tables.

Once the data file is correctly created, it is necessary to locate its position in the computer, meaning to know what is called the "absolute path" of the file.

In Windows, the absolute path is indicated by the right-click of the mouse in the properties of the file, to "Location". Let us take the example with the maize.txt file (download it at https://c3bi.pasteur.fr/gael-millot-livres/ and unzip it). When placed on the Windows desktop, the absolute path of this file on my computer is: C:\Users\Gael\Desktop\maize.txt.

For Mac OS, the absolute path is indicated by Ctrl + mouse click in "Read Information" to "Location". Same example as for Windows: when the file is placed on the Mac OS desktop, the absolute path of this file in my computer is: /Users/Gael/Desktop/maize.txt.

Notes:

1. R uses slash (symbol "/") or double backslash (symbol "\\") as delimiter in the absolute path of a file, without space or carriage return ("Enter" key on the keyboard) before or after
the delimiter. For Windows, it is necessary to replace each backslash "\" of the path displayed by a slash or a double backslash.

(2) This example is practical, but with experience, you should get used to store your files in specific folders, and thus to exploit the new absolute paths.

(3) The foreign package allows compatibility of R with data from other statistical software.

Creation of data frames using the read.table() function

Detail of the main arguments of the function:

- **header**: indicates that the first row of the imported table is the column names (default header = FALSE)
- **sep**: specifies the delimiter of the imported table (takes spaces, tabs, carriage return, etc. by default)
- **dec**: indicates the character used as the decimal separator in the imported table (default dec = ".")
- **row.names**: assigns row names
- **col.names**: assigns column names
- **as.is**: controls the conversion of data during importation. By default, character columns are converted into factors. See also the stringsAsFactors argument in the help of the function
- **na.strings**: indicates the string that correspond to missing data (default is na.strings = "NA")
- **skip**: specifies the number of rows to remove before importing the table
- **nrows**: specifies the maximum number of rows from the table to import

To create a data frame object called maize from the file maize.txt:

```r
> maize<-read.table("C:/Users/Gael/Desktop/maize.txt", header=TRUE) # modify the character string highlighted in grey if the path is different. Do not to put space or carriage return before or after the slashes
> maize

Individual Height Mass Nb.grains Mass.grains Colour Ear.germination
1 1 NA NA NA <NA> <NA>
2 2 199 1431 320 92.1 Red No
3 3 205 1468 290 89.4 Yellow No
... ...
98 98 259 1324 199 51.9 Red No
99 99 268 1903 422 128.1 Yellow No
100 100 269 1722 333 101.4 Yellow.rouge No
```

The other columns (variables "Rooting" to "Right.censored") are not shown here but are displayed in RStudio:
The `header = TRUE` argument indicates that the first row of the imported file is the column names. Thus, be careful with this argument (default `header = FALSE`) depending on whether there are column names or not. Example:

```r
> maize<-read.table("C:/Users/Gael/Desktop/maize.txt") # modify the character string highlighted in grey if the path is different
> maize
```

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>V5</th>
<th>V6</th>
<th>V7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual</td>
<td>Height</td>
<td>Mass</td>
<td>Nb.grains</td>
<td>Mass.grains</td>
<td>Colour</td>
<td>Ear.germination</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>199</td>
<td>1431</td>
<td>320</td>
<td>92.1</td>
<td>Red</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>99</td>
<td>98</td>
<td>259</td>
<td>1324</td>
<td>199</td>
<td>51.9</td>
<td>Red</td>
</tr>
<tr>
<td>100</td>
<td>99</td>
<td>268</td>
<td>1903</td>
<td>422</td>
<td>128.1</td>
<td>Yellow</td>
</tr>
<tr>
<td>101</td>
<td>100</td>
<td>269</td>
<td>1722</td>
<td>333</td>
<td>101.4</td>
<td>Yellow.rouge</td>
</tr>
</tbody>
</table>

Without `header = TRUE`, column names are assigned by default ("V" for "Variable"). In this case, the actual column names are considered to be the measurements made on the first individual. The object imported this way cannot be used for statistical analyzes.

Note: If the first column does not have a name (e.g., deletion of "Individual" in the first line of the file `maize.txt`) then the function `read.table()` uses this column to name the rows of the data frame.
6.2. Description of data objects

A data object is characterized by (1) the number of elements included and (2) the nature of these elements. The classical functions to describe a data object are `length()`, `mode()`, `class()` and `summary()`.

6.2.1. The `length()` function

The function gives different information depending on the type of data object:

<table>
<thead>
<tr>
<th>Kind of data object</th>
<th>Description using the length() function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>Number of elements</td>
</tr>
<tr>
<td>Matrix</td>
<td>Number of cases</td>
</tr>
<tr>
<td>Array</td>
<td>Number of cases</td>
</tr>
<tr>
<td>Data frame</td>
<td>Number of columns</td>
</tr>
<tr>
<td>List</td>
<td>Number of compartments</td>
</tr>
<tr>
<td>Factor</td>
<td>Number of elements</td>
</tr>
<tr>
<td>Table</td>
<td>Number of cases</td>
</tr>
</tbody>
</table>

If all the objects from paragraph 6.1.1 have been created, here is an example of the information given by the `length()` function:

```r
> length(x)
[1] 1
> length(height)
[1] 3
> length(matrix.1)
[1] 15
> length(array.1)
[1] 30
> length(dataframe.1)
[1] 5
> length(list.1)
[1] 4
> length(factor.1)
[1] 6
> length(table.2)
[1] 6
```

6.2.2. The `mode()` function

The mode defines the nature of the elements of an object, which are obtained by the `mode()` function. The most common are the numeric, character, and logical modes. The numeric mode refers to objects composed of numbers, on which arithmetic operations can be performed (addition, mean, standard deviation, etc.). The character mode indicates that the object is made of strings. Each element of the object is enclosed between quotes. Even if the character object is only made of numbers, it cannot be used for arithmetic operations. The logical mode means that the object contains only the special features `TRUE`, `FALSE` and `NA`.

Warning: the `c()`, `matrix()` et `array()` functions require that the elements, that have to be inserted in such object, present the same mode. If R detects any problem, it uses the "character" mode, which can be a source of bugs if the data are initially numeric because some statistical functions of R will refuse to process these data. The `data.frames()` and `list()` functions accept multiple modes.
If all the objects from paragraph 6.1.1 have been created, here is an example of information given by the `mode()` function:

```r
> mode(x)
[1] "numeric"
> mode(height)
[1] "numeric"
> mode(logical.1)
[1] "logical"
> mode(matrix.1)
[1] "numeric"
> mode(array.1)
[1] "character"
> mode(dataframe.1)
[1] "list"
> mode(list.1)
[1] "list"
> mode(factor.1)
[1] "numeric"
> mode(table.2)
[1] "numeric"
```

Four remarks:

1. Since data frames and lists can contain multiple modes, the function returns "list", even if the object is made of elements of the same mode.

2. Be careful when creating objects of character mode: character strings must be enclosed between quotes. If the quotes are not present, the characters are considered by R as object names, which generates either a bug (the object does not exist and thus is not recognized) or generates a character vector but not the expected one. In addition, numbers between quotes are considered as characters. Arithmetic operations will no longer be performed on these values. Examples:

   ```r
   > c(notgood, thisvectoris)
   Error: objet 'notgood' not found
   > c("colour", "rooting")
   [1] "colour" "rooting"
   > c(colour, rooting)  # the elements of the color object and the rooting object are joined
   [1] "Red" "Yellow" "Yellow" "Weak" "Intermediate" "Intermediate" "Intermediate" "Strong" "Strong"
   > 2*c(130,300,170)  # multiplication by 2 possible
   [1] 260 600 340
   > 2*c("130","300","170")  # multiplication impossible on characters
   Error in 2 * c("130", "300", "170"): non-numeric argument to binary operator
   ```

3. The factor object is a special object since factor.1 displays characters while `mode(factor.1)` signals a numeric mode object. In fact, the factor systematically records all of its elements as numbers (see paragraph 6.1.1). However, arithmetic operations cannot be performed on a factor.

4. The `mode()` function, like the `class()` function below, does not distinguish between an integer and a double-precision digit (see the help of the `double()` function).
6.2.3. The `class()` function

The function indicates the nature of the object except if it is a vector. In such situation, it indicates, most of the time, the mode. If all the objects from paragraph 6.1.1 have been created, here is an example of the information given by the `class()` function:

```r
> class(height)
[1] "numeric"
> class(colour)
[1] "character"
> class(logical.1)
[1] "logical"
> class(matrix.1)
[1] "matrix"
> class(array.1)
[1] "array"
> class(dataframe.2)
[1] "data.frame"
> class(list.1)
[1] "list"
> class(factor.1)
[1] "factor"
> class(table.2)
[1] "table"
```

6.2.4. The `summary()` function

This function provides a sum of practical information, which varies according to the data objects. Results are summarized in the following three tables:

<table>
<thead>
<tr>
<th>Code used in the next tables</th>
<th>Kind of result provided by the <code>summary()</code> function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Minimum, maximum, mean and quartiles</td>
</tr>
<tr>
<td>B</td>
<td>Number, class and mode</td>
</tr>
<tr>
<td>C</td>
<td>Mode and number of FALSE and TRUE</td>
</tr>
<tr>
<td>D</td>
<td>A for each column</td>
</tr>
<tr>
<td>E</td>
<td>Contingency in each column</td>
</tr>
<tr>
<td>F</td>
<td>C for each column</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kind of data object</th>
<th>Result of <code>summary()</code> depending on the mode of the object</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Numeric</td>
</tr>
<tr>
<td>Vector</td>
<td>A</td>
</tr>
<tr>
<td>Matrix</td>
<td>D</td>
</tr>
<tr>
<td>Array</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kind of data object</th>
<th>Result of <code>summary()</code> depending on the object class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data frame</td>
<td>A for the numeric variables, C for logical variables and class frequencies for factors</td>
</tr>
<tr>
<td>List</td>
<td>B for each compartment of the list</td>
</tr>
<tr>
<td>Factor</td>
<td>Frequencies of the different classes (counting)</td>
</tr>
<tr>
<td>Table</td>
<td>Number of cases, number of factors and a χ² test</td>
</tr>
</tbody>
</table>
If all the objects from paragraph 6.1.1 have been created, here is an example of information given by the `summary()` function:

```r
> summary(height)
     Min.   1st Qu.    Median      Mean   3rd Qu.    Max.
      130       150       170       200       235       300

> summary(colour)
     Length   Class     Mode
         3  character  character

> summary(logical.1)
     Mode   FALSE    TRUE   NA's
        logical       1       2       0

> summary(matrix.1)
     V1            V2            V3
    Min.   :1   Min.   : 6   Min.   :11
  1st Qu.:2   1st Qu.: 7   1st Qu.:12
  Median  :3   Median  : 8   Median  :13
  Mean    :3   Mean    : 8   Mean    :13
  3rd Qu.:4   3rd Qu.: 9   3rd Qu.:14
  Max.    :5   Max.    :10   Max.    :15

> summary(array.1)
     Length   Class     Mode
         30  array  character

> summary(dataframe.2)
     height   mass  nb.grains    colour
     Min.    130   Min.   :1398 Yellow:2
  1st Qu.:150  1st Qu.:1415 Red:1
  Median :170 Median :1431 Median :290.0
  Mean :200 Mean :1432 Mean :252.3
  3rd Qu.:235  3rd Qu.:1450 3rd Qu.:305.0
 Max. :300 Max. :1468 Max. :320.0

> summary(list.1)
     Length  Class     Mode
       [1,]    1   -none- numeric
       [2,]    3   -none- numeric
       [3,]   15   -none- numeric
       [4,]    4 data.frame list

> summary(factor.1)
     Weak   Strong Intermediate
        1      2      3

> summary(table.2)
Number of cases in table: 6
Number of factors: 2
Test for independence of all factors:
  Chisq = 0.75, df = 2, p-value = 0.6873
  Chi-squared approximation may be incorrect

6.3. Analysis and modification of data in a data objects

Procedures are not given for the matrices, arrays and tables. They are manipulated like the data frames, with potential additional dimensions to consider. Warning: In R, a data object modification is generally apparent (nondestructive), meaning that the modification is displayed on the screen but it does not affect the object in question. To save a modification in an object, assignation has to be used. By taking the first example below: height/2 does not modify the height vector compared to height <- height/2.
Vectors

Let us take the height and rooting vectors in paragraph 6.1.1, created as follows:

```
> height<-c(130,300,170)
> rooting<-rep(c("Weak", "Intermediate", "Strong"), c(1,3,2))
```

Division of a vector:
```
> height/2
[1]  65 150  85
```

Vector analysis:
```
> height<200
[1] TRUE FALSE TRUE
```

Extraction of the 3rd element of a vector:
```
> height[3]
[1] 170
```

Extraction of the 3rd and 5th elements of a vector:
```
> rooting[c(3,5)]
[1] "Intermediate" "Strong"
```

Extraction of all the elements except the first:
```
> height[-1]
[1] 300 170
```

Extraction of the elements greater than the value 150:
```
> height[height>150]
[1] 300 170
```

Extraction of the elements between 100 and 200:
```
> height[height>=100 & height<=200]
[1] 130 170
```

Extraction of the two smallest elements:
```
> sort(height)[1:2]  # sort(height) sorts the items in ascending order (default) and
[1:2] takes the first and second element of the sorted height
[1] 130 170
```

To find the location of an element in a vector:
```
> which(height==300)
[1] 2

> which(rooting=="Strong")
[1] 5 6
```

The second element of the height vector is the value 300. The same with the fifth and sixth

To replace one value of a vector by another one:
```
> height.2<-height  # height.2 is used instead of height so that the height vector
                     # is not modified and will serve later
> height.2[which(height.2==300)]<-1000  # which(height.2==300) produces the result
2. Therefore, the instruction is equivalent to height.2[2]<-1000
> height.2
[1] 130 1000 170
```

To name the elements of a vector:
```
> names(height) <- c("a", "b", "c")  # or directly during a vector creation:
      height<-c(a = 130, b = 300, c = 170)
> height
```
To delete names in a vector:
```r
> names(height) <- NULL
> height
130 300 170
```

Matrices and arrays

Matrices and arrays are somehow handled like data frames (see below).

Data frames

Let us use the data frame `dataframe.2` created in paragraph 6.1.1 as follows:
```r
> height<-c(130,300,170)
> mass<-c(1431,1468,1398)
> nb.grains<-c(320,290,147)
> colour<-c("Red","Yellow","Yellow")
> dataframe.2<-data.frame(height, mass, nb.grains, colour)
```

To know the number of rows and columns in a data frame:
```r
> dim(dataframe.2) # gives the number of rows, then columns
[1] 3 4
```

To know the variable names (the name of the columns):
```r
> names(dataframe.2) # or colnames(dataframe.2)
[1] "height" "mass" "nb.grains" "colour"
```

To know the name of column 2:
```r
> names(dataframe.2)[2] # or names(dataframe.2)[[2]]
[1] "mass"
```

To know the row and column names of a data frame:
```r
> dimnames(dataframe.2)
[[1]]
[1] "1" "2" "3"
[[2]]
[1] "height" "mass" "nb.grains" "colour"
```

The result is a "list" object with [[1]] for rows and [[2]] for columns.

To change the name of column 2:
```r
> dataframe.5 <- dataframe.2 # dataframe.5 is the same as dataframe.2. It is used for procedures that modify a data frame. Thus, dataframe.2 remains intact and will serve later
> names(dataframe.5)[2]<-"total.mass" # or dimnames(dataframe.5)[[2]][2]<-"total.mass"
> dataframe.5

height total.mass nb.grains colour
1 130 1431 320 Red
2 300 1468 290 Yellow
3 170 1398 147 Yellow
```

To change the name of row 2:
```r
> row.names(dataframe.5)[2]<-"Bob" # or dimnames(dataframe.5)[[1]][2]<-"Bob"
> dataframe.5

height total.mass nb.grains colour
1 130 1431 320 Red
Bob 300 1468 290 Yellow
3 170 1398 147 Yellow
```
To delete row names:

```r
> dataframe.6 <- dataframe.5  # to keep the row names in dataframe.5
> row.names(dataframe.6) <- NULL
> dataframe.6
  height total.mass nb.grains colour
1     130         1431       320   Red
2     300         1468       290   Yellow
3     170         1398       147   Yellow
```

To delete column names:

```r
> names(dataframe.6) <- NULL
> dataframe.6
  NA   NA  NA    NA
1 130 1431 320 Red
2 300 1468 290 Yellow
3 170 1398 147 Yellow
```

To change all the column names of a data frame:

```r
> names(dataframe.6) <- c("a","b","c","d")
> dataframe.6
  a    b   c     d
1 130 1431 320 Red
2 300 1468 290 Yellow
3 170 1398 147 Yellow
```

Warning: some column names are replaced by NA if the number of elements in the vector is less than the number of columns. These NAs are painful because the elements in the NA columns are no longer identified by certain functions. It is important to avoid this situation.

Extract the element from row 1 and column 2 using the row number and the column name:

```r
> dataframe.2[1,"mass"]
[1] NA
```

The data frame uses the same position designation as the matrices. The number before the comma within square brackets indicates the row number and the number after the comma the column number. You can use names or numbers, for both rows and columns.

To extract the 3\textsuperscript{rd} variable (column 3) from the data frame (result of class "data frame"):

```r
> dataframe.2[3]  # or dataframe.2["nb.grains"]
  nb.grains
1 320
2 290
3 147
```

To extract the 3\textsuperscript{rd} variable (column 3) from the data frame (result of class "vector"):

```r
> dataframe.2[,3]  # or dataframe.2[,"nb.grains"]
[1] 320 290 147
```

To extract a row from the data frame (result of class "data frame"):

```r
> dataframe.5[2,]  # or dataframe.5["Bob",]
  height total.mass nb.grains colour
Bob 300 1468 290 Yellow
```

Another specific technique for data frames and lists. The variable (column) in a data frame can be specified using the "$" symbol:

```r
> dataframe.2$height  # equivalent to dataframe.2[,1] (result of class "vector")
[1] 130 300 170
```
It is not necessary to write the full column name. Only the first characters can be specified, provided that this does not correspond to several variables of the data frame. Example with the variable "Height":

```r
> dataframe.2$h
[1] 130 300 170
```

If the required variable does not exist or if the characters used after "$" correspond to several variables in the data frame, R returns "NULL". Be careful though: diminutives cannot be used when modifying a data frame column (see below). Finally, avoiding diminutives is a good computing practice.

To extract elements 2 and 3 from variable 1:

```r
> dataframe.2$height[c(2,3)] # or dataframe.2[,1][c(2,3)]. However,
  dataframe.2[1][c(2,3)] does not work and produces an error message, but
  dataframe.2[1][c(2,3), 1] works
[1] 300 170
```

To extract the values greater than 150 and less than or equal to 250, from the variable "Height":

```r
> dataframe.2$height[dataframe.2$height > 150 & dataframe.2$height <= 250] # or
  dataframe.2[,1][dataframe.2[,1] > 150 & dataframe.2[,1] <= 250]
[1] 170
```

Extract the values of the mass from the individuals (rows) whose height values are greater than 150:

```r
> dataframe.2$mass[dataframe.2$height > 150] # or dataframe.2[,2][dataframe.2[,1] >
  150]
[1] 1468 1398
```

Extract a subset of a data frame:

```r
> dataframe.2[dataframe.2$height > 150, ] # or dataframe.2[dataframe.2[,1] > 150,]
height mass nb.grains colour
2     300  1468       290   Yellow
3     170  1398       147   Yellow
```

The comma behind the value 150 indicates that the search is performed on rows. The rows for which the height values are greater than 150 are extracted.

Can also be written:

```r
> subset(dataframe.2, height > 150)
# height is the name of the data frame column on which to perform the selection
```

To find the location of an item by its value:

```r
> which(dataframe.2=="Red" | dataframe.2==300, arr.ind=TRUE)
  row col
 [1,] 2 1
 [2,] 1 4
```

The operator | means or. The result indicates the row and column number for each item that matches the query. If the arr.ind argument is not used, the given value is the sequence number in the data frame when it is filled in by column starting from topleft.

To know the nature of the columns of a data frame:

```r
> sapply(dataframe.2, class)
  height mass nb.grains colour
"numeric" "numeric" "numeric" "factor"
```

To change the column 2 of a data frame:

```r
> dataframe.5$height<-dataframe.5$height-50 # warning: diminutive name of height
  cannot be used during assignment. The expression dataframe.5$h <-
  dataframe.5$height-50 creates a new column named h in dataframe.5. It is also
```
possible to use `dataframe.5 <- transform(dataframe.5, height=height-50)` but be careful with the `transform()` function: the `=` sign cannot be replaced by the `<-` assignment operator.

```r
> dataframe.5
table的高度 每个星期的总重量 粒数 颜色
1 80 1431 320 Red
Bob 250 1468 290 Yellow
3 120 1398 147 Yellow
```

Modification of the first two numerical values of the 1st column:

```r
> dataframe.5$height[c(1,2)]<-c(1000,2000)
> dataframe.5
table的高度 每个星期的总重量 粒数 颜色 height - 100
1 1000 1431 320 Red 30
Bob 2000 1468 290 Yellow 200
3 120 1398 147 Yellow 70
```

To add a column in a data frame:

```r
> cbind(dataframe.5, height-100)
# warning: written this way, height corresponds to the previously created vector, not to the data frame column (see the following example). Reminder: a result is a simple display. To modify dataframe.5, the following assignment must be made: dataframe.5 <- cbind(dataframe.5, height-100). In addition, the name of the new column can be specified directly:
> cbind(dataframe.5, Final.height = height-100)
height 每个星期的总重量 粒数 颜色 height - 100
1 1000 1431 320 Red 30
Bob 2000 1468 290 Yellow 200
3 120 1398 147 Yellow 70
```

The `transform()` function easily adds a column by modifying a column already present:

```r
> transform(dataframe.5, subtraction=height-100)
# compare this result with the result of the previous example
height 每个星期的总重量 粒数 颜色 subtraction
1 1000 1431 320 Red 900
Bob 2000 1468 290 Yellow 1900
3 120 1398 147 Yellow 20
```

Another way to add a column in a data frame:

```r
> dataframe.7<-dataframe.5
> dataframe.7$Parcel<-c("South", "North", "South")
> dataframe.7
height 每个星期的总重量 粒数 颜色 Parcel
1 1000 1431 320 Red South
Bob 2000 1468 290 Yellow North
3 120 1398 147 Yellow South
```

To add a row in a data frame:

```r
> rbind(dataframe.5, c("a","a","a","a"))
# try with rbind(dataframe.5, c("a","a","a","Red")) to suppress the warning message and rbind(dataframe.5, c("a", "Red")) to see the recycling effect of a row filling height 每个星期的总重量 粒数 颜色
1 1000 1431 320 Red
Bob 2000 1468 290 Yellow
3 120 1398 147 Yellow
4 a a a <NA>
Warning message: In `rbind("*tmp*", ri, value = "a")`: incorrect factor level, NAs generated
```
Note that even if we cannot see it, all the numeric columns are modified into character mode because there can be only one mode in a data frame column. Thus, the numeric mode cannot be maintained here. In addition, a warning message is generated because the last "factor" column can only be filled with values that are already present, i.e. "Red" or "Yellow" only.

Unlike columns, the number of elements in the added row can be of any length. If it is less than the number of columns in the data frame, the additional columns are filled by recycling the values of the added row. If it is greater, only the first elements are used.

The `rbind()` and `cbind()` functions merge two data frames, showing the same number of columns and rows respectively (if one of the data frames is a matrix, the final result is a data frame). Concerning `cbind()`, if the two data frames have row names, the function takes the names of the first data frame specified in the function. If only one of the two data frames has row names, these are used regardless of the order of the data frames specified in the function. Concerning `rbind()`, the two data frames must have the same column names. These considerations remain valid if one of the data frames is a matrix.

The `rbind.data.frame()` and `cbind.data.frame()` functions are designed for matrix fusion. These functions are identical to `rbind()` and `cbind()` respectively, but convert the result into data frame.

To transform a data frame into a list:

```
> c(dataframe.2)
$height
[1] 130 300 170
$mass
[1] 1431 1468 1398
$nb.grains
[1] 320 290 147
$colour
[1] Red Yellow Yellow
Levels: Yellow Red
```

Lists

Let us use the list `list.1` created in paragraph 6.1.1 as follows:

```
> x<-4
> height<-c(130,300,170)
> matrix.1<-matrix(1:15,ncol=3)
> mass<-c(1431,1468,1398)
> nb.grains<-c(320,290,147)
> colour<-c("Red","Yellow","Yellow")
> dataframe.2<-data.frame(height, mass, nb.grains, colour)
> list.1<-list(x, height, matrix.1, dataframe.2)

To name the compartments of a list:

```
> names(list.1)<-c("a","b","c","d") # or directly during the creation: list.1<-
    list(a = x, b = height, c = matrix.1, d = dataframe.2)
> list.1
$a
[1] 4

$b
[1] 130 300 170
```
To extract the second compartment from the list (result of class vector, matrix, etc., if the object initially present was a vector, a matrix, etc.):

```r
> list.1[[2]] # or list.1$b
[1] 130 300 170
```

To extract the second compartment from the list (result of class list):

```r
> list.1[2] # list.1[[c(1,4)]] to get the compartments 1 et 4
$b
[1] 130 300 170
```

To extract or join list compartments:

```r
> c(list.1[2], list.1[4])
$b
[1] 130 300 170
```

To extract the 2nd element in the 3rd column of the 4th compartment:

```r
> list.1[[4]][,3][2] # or list.1$d$nb.grains[2]
[1] 290
```

To find the location of an element using its value:

```r
> which(list.1[[3]]==9,arr.ind=TRUE) # which(list.1$c==9,arr.ind=TRUE)
      row col
[1,]   4   2
```

The `which()` function is only applicable on a predefined compartment, not on the entire list (`which(list.1 == 6)` does not work, we have to use `lapply(lapply( list.1, "==", 6), which)`).

To clear the names of the compartments of a list:

```r
> names(list.1)<-NULL
> list.1
[[1]]
[1] 4

[[2]]
[1] 130 300 170

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

---

$$c$$

<table>
<thead>
<tr>
<th></th>
<th>[,1]</th>
<th>[,2]</th>
<th>[,3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

$$d$$

<table>
<thead>
<tr>
<th>height</th>
<th>mass</th>
<th>nb.grains</th>
<th>colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>130</td>
<td>1431</td>
<td>320</td>
</tr>
<tr>
<td>2</td>
<td>300</td>
<td>1468</td>
<td>290</td>
</tr>
<tr>
<td>3</td>
<td>170</td>
<td>1398</td>
<td>147</td>
</tr>
</tbody>
</table>

To extract the second compartment from the list (result of class vector, matrix, etc., if the object initially present was a vector, a matrix, etc.):

```r
> list.1[[2]] # or list.1$b
[1] 130 300 170
```

To extract the second compartment from the list (result of class list):

```r
> list.1[2] # list.1[[c(1,4)]] to get the compartments 1 et 4
$b
[1] 130 300 170
```

To extract or join list compartments:

```r
> c(list.1[2], list.1[4])
$b
[1] 130 300 170
```

To extract the 2nd element in the 3rd column of the 4th compartment:

```r
> list.1[[4]][,3][2] # or list.1$d$nb.grains[2]
[1] 290
```

To find the location of an element using its value:

```r
> which(list.1[[3]]==9,arr.ind=TRUE) # which(list.1$c==9,arr.ind=TRUE)
      row col
[1,]   4   2
```

The `which()` function is only applicable on a predefined compartment, not on the entire list (`which(list.1 == 6)` does not work, we have to use `lapply(lapply( list.1, "==", 6), which)`).

To clear the names of the compartments of a list:

```r
> names(list.1)<-NULL
> list.1
[[1]]
[1] 4

[[2]]
[1] 130 300 170

[[3]]
[,1] [,2] [,3]
[1,] 1  6  11
[2,] 2  7  12
[3,] 3  8  13
[4,] 4  9  14
[5,] 5 10  15
```
Factors

Let us use the factor `rooting.2` created in paragraph 6.1.1, as follows:

```r
> rooting.2 <- factor(c(5, 0, 10, 5, 40, 5, 0), levels = c(0, 5, 10, 40),
  labels = c("Weak", "Intermediate", "Strong", "Very.strong"))
```

To extract elements 3 and 5:

```r
> rooting.2[c(3,5)]
[1] Strong Very.strong
Levels: Weak Intermediate Strong Very.strong
```

To extract all the elements but the first:

```r
> rooting.2[-1]
Levels: Weak Intermediate Strong Very.strong
```

To find the position of an element:

```r
> which(rooting.2 == "Weak")
[1] 2 7
```

To display the levels of a factor:

```r
> levels(rooting.2)
[1] "Weak"   "Intermediate"  "Strong"  "Very.strong"
```

To display the 2nd level:

```r
> levels(rooting.2)[2]
[1] "Intermediate"
```

To display the numerical value of a factor:

```r
> as.numeric(rooting.2) # whatever numbers were used at the creation of a factor
  (see paragraph 6.1.1), the factor() function redefines them beginning by 1 and
  incrementing the numbers by 1 each time (e.g. 0 becomes 1, 5 becomes 2, 10
  becomes 3 and 40 becomes 4). In other words, each level is defined by its
  numeric position in Levels: Weak is the first, Intermediate the second, etc.
  The as.numeric() function does not work on character vectors, which
  differentiates them from factors
[1] 2 1 3 2 4 2 1
```

To sort the elements of the factor:

```r
> sort(rooting.2) # beware: the sort() function uses the order of the levels
  displayed in Levels, not the alphabetical order of these levels. Therefore,
  sort() behaves as if it uses the numeric values associated to each level
Levels: Weak Intermediate Strong Very.strong
```

To reorder the levels of a factor (shown in Levels):

```r
> rooting.3 <- relevel(rooting.2, ref = "Very.strong") # beware: the ref argument can
  only take a single existing level of the factor, then move it as the first
  level in the levels order, shown in Levels. But we can use relevel() as many
  times as needed to obtain the desired order of the levels. Alternatively, we
  can use the following procedure: rooting.3 <- factor(as.numeric(rooting.2),
  levels = c(4, 3, 2, 1), labels = c("Very.strong", "Strong", "Intermediate",
  "Weak"))
```

```r
> rooting.3
```

1 3 2 4 2 1
To reorder the levels of a factor (transformation into an "ordered" factor):

```
> rooting.4 <- ordered(rooting.3, levels = c("Weak", "Intermediate", "Strong", "Very.strong"))
```

# the ordered() function also allows the user to modify the names of the level. For example, ordered(rooting.3, labels = c("a", "b", "c", "d")): Very.strong becomes a, Strong becomes b, etc. It is also possible to perform both a renaming and a reordering of the levels at the same time. For example, with ordered(rooting.3, levels = c("Weak", "Intermediate", "Strong", "Very.strong"), labels = c("a", "b", "c", "d")): Weak becomes a, Intermediate becomes b, etc.

```
> rooting.4
Levels: Weak < Intermediate < Strong < Very.strong
```

In "Levels", the levels are now ordered.

To modify the 1st, 2nd et 4th levels of a factor:

```
> levels(rooting.3)[c(1, 2, 4)] <- "Unknown"
```

```
> rooting.3
```

Levels: Unknown Intermediate

To recover the alphabetical order of the levels:

```
> as.factor(as.character(rooting.3))
```

```
```

Levels: Intermediate Unknown

Tables

See the data frame section for the common manipulations of tables.

Let us use table.2 created in paragraph 6.1.1, as follows:

```
> rooting <- rep(c("Weak", "Intermediate", "Strong"), c(1, 3, 2))
> parcel <- c("North", "North", "North", "South", "North", "South")
> table.2 <- table(rooting, parcel)
```

To obtain the number of rows and columns of a table:

```
> dim(table.2) # outputs first the number of rows, then the number of columns
[1] 3 2
```

To obtain the total count:

```
> margin.table(table.2)
[1] 6
```

To obtain the counts by row:

```
> margin.table(table.2, 1) # also, try addmargins(table.2) then addmargins(table.2)[,3]
rooting
Weak   Strong Intermediate
 1      2      3
```

To obtain the counts by column:

```
> margin.table(table.2, 2) # also, try addmargins(table.2)[4,]
parcel
North  South
 4      2
```
To display the table with the sums (total, by row and by column):

```r
> addmargins(table.2)
```

<table>
<thead>
<tr>
<th>parcel</th>
<th>North</th>
<th>South</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Strong</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Intermediate</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Sum</td>
<td>4</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

6.5. Import instructions

There is an elegant way to provide instructions to R: the use of the `source()` function. A code written in the editor pane can be saved as follows:

R script will be saved as a "R" file, Text File as a "txt" file. Then, this file can be "sourced", which is the term usually used to import instructions in a console:

```r
> rm("colour", "dataframe.2", "height", "mass", "nb.grains") # to remove these objects if they are already present in RStudio
> source("C:/Users/Gael/Desktop/source_test.R") # or source("C:/Users/Gael/Desktop/source_test.txt"). Modify the character string highlighted in grey if the path is different
> ls() # the 5 objects are now present in the working environment
[1] "colour"  "dataframe.2"  "height"  "mass"  "nb.grains"
6.6. Export of data objects

Different kinds of functions are available, depending on the kind of object and the kind of file to obtain:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>write()</td>
<td>export an object into a file</td>
</tr>
<tr>
<td>write.table()</td>
<td>export a matrix or a data frame in a file</td>
</tr>
<tr>
<td>write.csv()</td>
<td>export a matrix or a data frame as a .csv file</td>
</tr>
<tr>
<td>dput()</td>
<td>export data in a file of ASCII type</td>
</tr>
</tbody>
</table>

Example:

```r
> dir.create("C:/Users/Gael/Desktop/result.file") # dir.create() creates a folder using the specified path. The last element of the path is the name of the created folder, which is placed in the penultimate element of the path. Warning: if the folder already exists, a warning message is displayed but the existing folder is not overwritten. Modify the character string highlighted in grey if the path is different
> dput(dataframe.2, file="C:/Users/Gael/Desktop/result.file/dataframe2.txt")
> write.table(dataframe.2, file="C:/Users/Gael/Desktop/result.file/dataframe2_bis.txt")
> write.csv(dataframe.2, file="C:/Users/Gael/Desktop/result.file/dataframe2.csv")
```

Beware with these functions: if the file already exists on the computer, it is overwritten without warning.

7. Manipulation of conditional or repeated instructions

These operations are performed by special features of R (see paragraph 4.4).

7.1. Conditional features of R

The if/else conditional features of R are used to test a condition and perform an instruction based on the true or false result of the condition being tested. Example: "if it is not noon yet, I go shopping, otherwise I stay at home". Depending on the time, the first or second instruction will be accomplished. Here are the different features of R:

<table>
<thead>
<tr>
<th>Conditional feature</th>
<th>Description</th>
<th>Possible to assign inside instruction?</th>
<th>Help command</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifelse()</td>
<td>ifelse(tested condition, instruction if the condition is true, instruction if the condition is false)</td>
<td>Yes using &lt;-</td>
<td>?ifelse</td>
</tr>
<tr>
<td>if(){else{}}</td>
<td>if(tested condition){one or several instructions if the condition is true}else{one or several instructions if the condition is false}</td>
<td>Yes using &lt;- or =</td>
<td>?&quot;if&quot;</td>
</tr>
</tbody>
</table>

Example with `ifelse()`:

```r
> i<-c(2,25)
> ifelse(i < 4, i^2, i^0.5) # if the value of i is strictly less than 4, the instruction "i to the power 2" is used, otherwise "i to the power 0.5 (meaning the square root)" is executed
[1] 4 5
```
With the value 2 of i, the condition is true. The feature then performs the first instruction and 4 is obtained. With the value 25, the condition is false. The feature performs the second instruction and 5 is obtained.

Note: This feature tests several times the condition i<4 simultaneously, which is not the case for the if(){else{}} feature. But be careful with ifelse(): we might think that the feature starts by testing the first value of i, then the second value, and so on. In fact, the evaluation is carried out without repetition loop. Example with the previous code by adding an assignment:

```r
> x<-NULL # create a NULL vector
> i<-c(2,25)
> ifelse(i < 4, x<-i^2, i^0.5) # if the value of i is strictly less than 4, the instruction "i to power 2" is assigned to x. The results of ifelse() do not change.
[1] 4 5
> x # On the other hand, we would expect that x contain the value 4 only. But this is not the case. Indeed, if one of the conditions is true (2<4 in our example), the whole vector i, composed of 2 and 25, is set to the power of two and is assigned to x
[1] 4 625
```

Another example with ifelse():

```r
> i<-c(2,25)
> ifelse(i == 2, i^2, NA) # use NA if no instruction is required
[1] 4 NA
```

With the value 25 of i, the condition is false. The feature does not perform any instruction and returns "NA".

Example with if(){else{}}:

```r
> i<-25 # i cannot have multiple elements with the if(){else{}} feature
> x<-NULL ; y<-NULL
> if(i < 4){x<-i^2 ; y<-100} else{x<-i^0.5 ; y<-55}
> x
[1] 5
> y
[1] 55
```

The condition is false. So x takes the square root of i and y takes the value 55.

It is possible to have only one condition to meet with if(){else{}}:

```r
> i<-25
> x<-NULL ; y<-NULL
> if(i < 4){x<-i^2 ; y<-100} # no obligation to test "else". Use if(i < 4){} else{x<-i^0.5 ; y<-55} if you want to use "else" only
> x
NULL
> y
NULL
```

Example of logical question:

```r
> i<-3
> if(is.numeric(i) == TRUE){i^2} # can also be written ifelse(is.numeric(i) == TRUE, i^2, NA)
[1] 9
```

Note: the is.numeric(i) function asks the question "Is it a numeric object?". The result of this function is of logical type (either TRUE or FALSE), which means that ==TRUE does not have to be written:

```r
> i<-3
> if(is.numeric(i)){i^2}
[1] 9
```
> is.numeric(i)
[1] TRUE

Warning: do not put a line break (carriage return) between the } brace of \code{if()} and the beginning of the \code{else{}} feature when writing the code.

7.2. Looping features of R

The common loops of type \code{for}, \code{while} and \code{repeat} exist in R:

<table>
<thead>
<tr>
<th>Looping feature</th>
<th>Number of repeats</th>
<th>Loop end</th>
<th>Breaking loop using</th>
<th>Help function</th>
</tr>
</thead>
<tbody>
<tr>
<td>\code{for{}}</td>
<td>specified between brackets</td>
<td>When the number specified between brackets is reached</td>
<td>Yes</td>
<td>&quot;for&quot;</td>
</tr>
<tr>
<td>\code{while{}}</td>
<td>Depending on a condition specified between brackets</td>
<td>When the tested condition is false</td>
<td>Yes</td>
<td>&quot;while&quot;</td>
</tr>
<tr>
<td>\code{repeat{}}</td>
<td>No limit</td>
<td>No end</td>
<td>Yes</td>
<td>&quot;repeat&quot;</td>
</tr>
</tbody>
</table>

Notes:
(1) In this document, "i" will be used by default in a \code{for} loop object, but any name can be used.
(2) Use the STOP icon in the top of the console pane to break the loop in progress, if necessary.
(3) The three kinds of loops can be interrupted by \code{break} or \code{stop()} written between the braces of the feature (see the example with the \code{repeat{}} feature below).
(4) Spaces are required before and after the feature \code{in} of \code{for{}} (see below).

The \code{for{}} feature repeats one or more statements (separated by semicolons) for a certain number of times, this number of times corresponding to the different values taken by an object created between the brackets. Example:

\begin{verbatim}
> i.value<-NULL ; x<-0
> for(i in 1:10){i.value<-c(i.value, i) ; x<-x+5}

# warning: the space before and after "in" is mandatory

> i
[1] 10
> i.value
[1]  1  2  3  4  5  6  7  8  9 10
> x
[1] 50
\end{verbatim}

The \code{i.value} object is empty at the beginning of the loop, and the \code{x} object is zero. The object \code{i} between the brackets takes the values 1 to 10. So the two instructions between braces are performed 10 times. At each repetition, the value of \code{i} is incorporated into the \code{i.value} vector and 5 is added to the value of \code{x}. At the end of the repetition loop, \code{i} possesses its last value, i.e. 10. The \code{i.value} vector has the 10 successive values taken by \code{i}, and \code{x} is 50.

The \code{for{}} loop uses the number of elements behind "in", not the nature of the elements:

\begin{verbatim}
> i.value<-NULL ; x<-0
> for(i in c(20,"1","chose")){i.value<-c(i.value, i); x<-x+5}

> i
[1] "chose"
> i.value
[1] "20" "1" "chose"
\end{verbatim}
Three repetitions were performed using this example.

The `while()` feature repeats one or more instructions (separated by semicolons), as long as the written conditions between the brackets are not fulfilled. We call this "conditional loop".

Example:
```r
> month.value<-NULL; x<-NULL
> month<-1
> while(month <= 5){
    month.value <- c(month.value, month) ;
    x<-c(x, month.name[month]) ;
    month<-month+1}
```

The "month" object must have an initial value before the loop is launched because it is on this object that the condition is tested, the condition being "Is the value of the "month" object less than or equal to 5"? Each time the condition is true, the instructions between braces are executed: (1) the value of "month" is included at the end of the "month.value" vector, (2) the element of the "month.name" object, at the position of the value of "month", is added to x and (3) "month" takes one more unit. At the end of the loop, "month" has the value 6 confirming that this value stopped the loop. The "month.value" vector shows the 5 successive values taken by "month" and x contains the first five elements of the "month.name" object.

8. R is a graphic editor

This paragraph addresses the basic concepts to know in order to plot graphs in R.

8.1. Discovery of the graphics window

Let's start with a simple example, the `plot()` function, one of the most commonly used functions, and draw \( y = x^2 \) depending on \( x \), with integer values ranging from 1 to 20:
```r
> x<-1:20
> plot(x, x^2)
```

The graph appears in the Plots tab of the tools pane (see paragraph 3.4.2.). It is important to identify the three different regions of the R graphic window: the plot region (which is the drawing region), the figure region (which includes the plot region and other chart components...
like axis, legends, titles, etc.), and the device region (which corresponds to the whole graphics window *per se*). By default, the figure region fills the entire graphics window. The two terms are thus, most of the time, equivalent. In paragraph 8.4, we will see how to change the relative size of these three regions.

### 8.2. Different kinds of graphics

#### 8.2.1. Description of the most used graphical functions

**R** has a range of functions enabling a multitude of graphical representations. Here are the most used for a single variable:

<table>
<thead>
<tr>
<th>Function</th>
<th>Kind of graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot()</td>
<td>Y-axis draw using different styles: dots, lines, bars, steps, etc.</td>
</tr>
<tr>
<td>hist()</td>
<td>Histogram</td>
</tr>
<tr>
<td>barplot()</td>
<td>Bar</td>
</tr>
<tr>
<td>barplot2()</td>
<td>Bar with intervals</td>
</tr>
<tr>
<td>stripchart()</td>
<td>One dimensional scatterplot</td>
</tr>
<tr>
<td>boxplot()</td>
<td>Box</td>
</tr>
<tr>
<td>qqnorm()</td>
<td>Quanlile-quantile plot based on the normal law</td>
</tr>
</tbody>
</table>

And here, the most used functions for the representation of two variables:

<table>
<thead>
<tr>
<th>Function</th>
<th>Kind of graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot()</td>
<td>Draw using different styles: dots, lines, bars, steps, etc.</td>
</tr>
<tr>
<td>qqplot()</td>
<td>Quantile-quantile plot</td>
</tr>
</tbody>
</table>

They are referenced in **R** as *high-level plotting commands* because they erase the potential existing graph and create a new one in the graphics window. As examples, we can use objects created in paragraph 6.1.1. Simply submit them to the desired function:

```r
> plot(maize$Height)
> hist(maize$Height)
> barplot(maize$Height)
> boxplot(maize$Height)
> stripchart(maize$Height)
> plot(maize$Height, maize$Mass)
```

The list of arguments of these functions is obtained by using the help function (`?plot()` for the `plot()` function for instance).

#### 8.2.2. Common arguments of these functions

An argument of a function is an option that, in the case of graphics functions, offers the possibility to improve the displayed graph. Most of the functions described in the previous paragraph have arguments in common:

- `xlim`: lower and upper limits on the abscissa axis (x-axis). Note: with the `boxplot()` function, `xlim` is not referring to the x-axis, but to the axis which separates the boxes.
**ᵧlim** lower and upper limits on the ordinate axis (y-axis). Note: with the `boxplot()` function, `ylim` is not referring to the y-axis but the axis parallel to the boxes.

**xlab** label (legend) of the x-axis.

**ylab** label (legend) of the y-axis.

**main** main title of the graph.

**sub** subtitle of the graph (displayed at the bottom of the graphical window, under the legend of the x-axis).

**cex** number indicating the amount by which plotting text and symbols should be magnified relative to the default (1 by default).

**cex.axis** number indicating the amount by which the size of axis annotation should be magnified relative to the default (1 by default, this value is multiplied by the value of `cex`).

**cex.lab** number indicating the amount by which the size of legends of axis should be magnified relative to the default (1 by default, this value is multiplied by the value of `cex`).

**cex.main** number indicating the amount by which the size of the main title should be magnified relative to the default (1 by default, this value is multiplied by the value of `cex`).

**cex.sub** number indicating the amount by which the size of the subtitle should be magnified relative to the default (1 by default, this value is multiplied by the value of `cex`).

**col.axis** color of the axis values (see paragraph 8.5)

**col.lab** color of the axis labels (see paragraph 8.5)

**col.main** color of the main title (see paragraph 8.5)

**col.sub** color of the subtitle (see paragraph 8.5)

**font** font to use for the text. Not used for the main title, the legends of axis and the subtitle

**font.axis** font to use for the axis values

**font.lab** font to use for the axis labels

**font.main** font to use for the main title

**font.sub** font to use for the subtitle

**bty** type of box drawn around the plot region.

**pch** type of plotted points (see `?points()`, and more particularly the example section, to determine which value of `pch` is associated to a particular form of point)

Let us take the example of plotting in paragraph 8.1 by using these arguments:

```r
> x<-1:20
> plot(x, x^2, xlim=c(0, 30), ylim=c(-100, 500), xlab="Variable x", ylab="Variable x squared", main="Square of the value 1 to 20", cex.axis=1.5, cex.lab=1.5, cex.main=2, bty="l", pch=16) # In bty="l", it is l as in the first letter of Lucy, not number 1
```
Note that for decreasing axes, simply reverse the values in `xlim` or `ylim`. Furthermore, do not hesitate to stretch the graphics window using the mouse if the scales or descriptions do not appear.

Warning: in general, the use of the "Enter" key within the caption text or title will create a new line when displayed in the graph. The best is to avoid this keyboard key at this time, and to use `\n` if necessary (write `?Quotes` in R for more information).

Some functions, such as `stripchart()`, cannot use all the arguments described above. In this case, it is always possible to find an equivalent way to modify graphs, addressed in paragraph 8.4.

### 8.3. Adding items to a graph

Here is a list of the most used functions to add items:

<table>
<thead>
<tr>
<th>Function</th>
<th>Add on the graph</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>points()</code></td>
<td>points</td>
</tr>
<tr>
<td><code>lines()</code></td>
<td>points joined by a line</td>
</tr>
<tr>
<td><code>abline()</code></td>
<td>a straight line without limits</td>
</tr>
<tr>
<td><code>segments()</code></td>
<td>a line segment</td>
</tr>
<tr>
<td><code>arrows()</code></td>
<td>an arrow</td>
</tr>
<tr>
<td><code>rect()</code></td>
<td>a rectangle</td>
</tr>
<tr>
<td><code>polygon()</code></td>
<td>a polygone</td>
</tr>
<tr>
<td><code>legend()</code></td>
<td>a legend</td>
</tr>
<tr>
<td><code>rug()</code></td>
<td>secondary ticks on axes</td>
</tr>
<tr>
<td><code>axis()</code></td>
<td>an axis on one side of the graph (if initially removed)</td>
</tr>
<tr>
<td><code>text()</code></td>
<td>text</td>
</tr>
<tr>
<td><code>mtext()</code></td>
<td>text in the margins of the figure or graphic windows</td>
</tr>
<tr>
<td><code>title()</code></td>
<td>text in the title, subtitle, legend or legends of the axes</td>
</tr>
</tbody>
</table>

Unlike functions of paragraph 8.2, they do not create a new plot in the graphics window when executed. This is why they are referred in R to as `low-level plotting commands`.  

![Graph of square of the value 1 to 20](image)
8.4. Graphical parameters

A graphic window is defined by 72 input parameters. Close the graphic windows which are opened in R and write the following instructions:

```r
> # Close all graphic windows (i.e. use the crossed icon of RStudio until not available
> par()
```

The number of parameters with the `par()` function can be obtained with the instruction `length(par())`. With `par(no.readonly=TRUE)`, only 66 parameters are displayed. Indeed, the 6 parameters `cin`, `cra`, `csi`, `cxy`, `din` and `page` can be found in the help of the `par()` function but cannot be modified.

The list of the 72 parameters is displayed in the console with their corresponding default values. By modifying these parameters, a plot can be greatly improved.

The graphical parameters are arguments of the `par()` function. Here is a list of the most used ones (see the help of the `par()` function for the full and detailed list of these parameters):

- **cex**: number indicating the amount by which plotting text and symbols should be magnified relative to the default (1 by default)
- **cex.axis**: number indicating the amount by which the size of axis annotation should be magnified relative to the default (1 by default, this value is multiplied by the value of `cex`)
- **cex.lab**: number indicating the amount by which the size of legends of axis should be magnified relative to the default (1 by default, this value is multiplied by the value of `cex`)
- **cex.main**: number indicating the amount by which the size of the main title should be magnified relative to the default (1 by default, this value is multiplied by the value of `cex`)
- **cex.sub**: number indicating the amount by which the size of the subtitle should be magnified relative to the default (1 by default, this value is multiplied by the value of `cex`)
- **col.axis**: color of the axis values (see paragraph 8.5)
- **col.lab**: color of the axis labels (see paragraph 8.5)
- **col.main**: color of the main title (see paragraph 8.5)
- **col.sub**: color of the subtitle (see paragraph 8.5)
- **font**: font to use for the text. Not used for the main title, the legends of axis and the subtitle. 1=plain, 2=bold, 3=italic, 4=bold italic, 5=symbol.
- **font.axis**: font to use for the axis values
- **font.lab**: font to use for the axis labels
- **font.main**: font to use for the main title
- **font.sub**: font to use for the subtitle
- **bty**: type of box drawn around the plot region.
- **lty**: type of the drawn line (line type)
- **lwd**: thickness of the drawn line (line width)
- **col**: changes the color of plotted items (see paragraph 8.5)
- **mar**: changes the width of the margins between the plot and the figure. By default `mar = c(5.1, 4.1, 4.1, 2.1)`. These four coordinates are attributed to the four sides of the plot, starting from the bottom and clockwise rotating. The values are in line numbers, which normally corresponds to 1/5 of inch (see `mai`)
- **mai**: same as `mar` but the values are in inches. By default `mar = c(1.02, 0.82, 0.82, 0.42)`
- **mgp**: change the distance relatively to the plot region, (1) of the axis legends, (2) of the axis values and (3) of the axis bar and tick marks. By default `mgp = c(3,1,0)`.
las

rotate axis values (but not the labels): las = 0 so that they are parallel to the axes (default), las = 1 so that they are horizontal, las = 2 so that they are perpendicular to the axes, las = 3 so that they are vertical.

mfrow

cut the figure region into several parts (mfrow = c(1,2) gives 1 row and 2 columns, resulting in two side-by-side figure regions). The plotted graphs are successively displayed according to rows. Warning: this is incompatible with the layout() function.

mfcol

same as mfrow except that the successive plotted graphs appear along the columns.

xpd

restrict the plots, texts, legends, etc., to the different regions of a graphics window (xpd = FALSE for the plot region, xpd = TRUE for the figure region, xpd = NA for the device region). If xpd = FALSE for instance, the added legend will be displayed only if it is positioned inside the plot region.

xaxt="n"

remove the x-axis (useful to subsequently customize the x-axis with the axis() function)

xaxt="s"

authorize the display of the x-axis (when xaxt = "n" was previously used).

yaxt="n"

remove the y-axis (useful to subsequently customize the y-axis with the axis() function)

yaxt="s"

authorize the display of the y-axis (when yaxt = "n" was previously used)

usr

shows the four values x_min, x_max, y_min and y_max of the plot region (axis boundaries). Use par(usr) or par("usr") after plotting the graph to obtain these values.

xaxp

inform on the graduations (tick marks) of the x-axis: the three displayed values x_min, x_max and n correspond respectively to the smallest, the largest and the number of intervals of the graduations. This facilitates the use of the rug() function that adds secondary tick marks on the x-axis. Use par(xaxp) or par("xaxp") after plotting the graph to obtain these values. See the help of par() in case of using logarithmic scale.

yaxp

same as xaxp for the y-axis.

plt

indicates the thickness of the margins between the plot region and the figure region. Values are the fractions (from 0 to 1) of the boundaries of the plot region, respectively left, right, bottom and top. This facilitates the positioning of the legend using the legend() function. Use par(plt) or par("plt") after plotting the graph to obtain these values.

xaxs="i"

prevent the fact that the x-axis exceeds the fixed limit by 4% (default). Thus, when using xlim = c(a_min, a_max) (see paragraph 8.2.2), the values x_min and x_max of the x-axis are extended from 4% to the right, i.e. x_min = a_min - 0.04 * (a_max - a_min), and 4% to the left, i.e. x_max = a_max + 0.04 * (a_max - a_min). Example: with xlim = c(0, 4), x_min = 0 - 0.04 * (4 - 0) = -0.16 and x_max = 4 + 0.04 * (4 - 0) = 4.16. The argument xaxs="i" remove this. In our example, x_min is set to 0 and x_max to 4.

yaxs="i"

prevent the fact that the y-axis exceeds the fixed limit by 4% (default). Same remarks as xaxs="i".

Note: modifications of graphical parameters are applied to the active graphics window. To be considered, they must be specified before the graph is plotted. Then, all graphics executed in the window will be affected by the modifications, even those of high level commands. To reset the parameters, the easiest way is to close all the graphics window, using the graphics.off() function.

Some arguments in par() are accepted by the graphics functions. Example: cex by plot().
But beware: in this case, they do not change the parameters of the graphics window. Any setting argument is temporary and only applies to the related high-level plotting command.

8.5. Colors

R has several functions for defining color settings in graphs:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col</td>
<td>graphical parameter which modifies the color of the drawn items (see paragraph 8.4)</td>
</tr>
<tr>
<td>colors()</td>
<td>list of 657 words used by the col parameter</td>
</tr>
<tr>
<td>palette()</td>
<td>link numbers to colors, which allows to use numbers instead of words in the col parameter</td>
</tr>
<tr>
<td>rgb()</td>
<td>set a color according to the primary colors (Red, Green, Blue)</td>
</tr>
<tr>
<td>gray()</td>
<td>set a grey levels</td>
</tr>
<tr>
<td>grey()</td>
<td>similar to gray()</td>
</tr>
<tr>
<td>hsv()</td>
<td>set a color according to Hue, Saturation and Value</td>
</tr>
</tbody>
</table>

Description of the colors() function

When you type `colors()` in R, 657 colors are displayed under the form of a list of words. To exploit these colors, just write the chosen words in the `col` argument of a graph function.

Example:

```r
> colors()
[1] "white"                "aliceblue"            "antiquewhite"
[4] "antiquewhite1"        "antiquewhite2"        "antiquewhite3"
[7] "antiquewhite4"        "aquamarine"           "aquamarine1"
...  
[649] "wheat3"               "wheat4"               "whitesmoke"
[652] "yellow"               "yellow1"              "yellow2"
[655] "yellow3"              "yellow4"              "yellowgreen"
```

> # Close all graphic windows
> par(ann=FALSE, xaxt="n", yaxt="n", bty="n") \# the argument ann manages the graphic annotations, meaning the display of axis labels and chart titles
> plot(1, col="aquamarine", pch=16, cex=10) \# "aquamarine" is the 8th word in the colors() list
Description of the `palette()` function

The `palette()` function assigns numbers to a set of colors. These can then be used through the `col` argument of a graphic function. By default, it associates values 1 to 8 with the following shades:

```r
> palette()
[1] "black"  "red"    "green3"  "blue"    "cyan"    "magenta" "yellow"
[8] "gray"
> plot(1, col=6, pch=16, cex=10)  # number 6 refers to the "magenta" color
```

Caution: If `plot(1, col=20, pch=16, cex=10)` is used while the color palette is not changed, the `col` argument assumes that the value of 20 matches "blue" (Counting is done by returning to the beginning of the list of eight words each time, which is equivalent to $2 \times 8 + 4 = 20$).

With `colors()`, it is not always easy to have a visual perception of the list of words displayed, and therefore to choose the colors you want. A technique using `palette()` solves this problem:

```r
> # Close all graphic windows
> palette(colors())  # assign numbers 1 to 657 for the 657 words in the list of colors()
> tempo<-NULL
> for(i in 14:1){tempo<-c(tempo, rep(i,50))}  # see paragraph 7.2
> par(ann=FALSE, xaxt="n", yaxt="n")
> plot(rep(1:50,14)[1:657], tempo[1:657], pch=22, bg=1:657, cex=1.5, bty="n")  # this time, it is not the argument col but the argument bg which uses the numbers of the palette
> par(xpd=TRUE)
> axis(side=2, at=14:1, labels=, cex.axis=1.5, srt=90)
> text(rep(-2, 14), 14:1, as.character((0:13)*50+1), srt=0, cex=1)
> text(c(10,20,30,40,50), rep(-0.5), c(10,20,30,40,50), srt=0, cex=1)
```
The graphic displays the 657 colors of colors() according to the numbers assigned by palette(). Numbers, indicated on the abscissa, facilitate counting. To choose a color, simply record its value and use the statement palette(colors()) before drawing the graph. Example with the color named "tomato":

```r
> # Close all graphic windows
> palette(colors())
> par(ann=FALSE, xaxt="n", yaxt="n", bty="n")
> plot(1, col=630, pch=16, cex=10) # you can also use plot(1, col="tomato", pch=16, cex=10) if palette(colors()) is not used
```

The statement palette("default") induces the return to the initial palette of 8 colors:

```
> palette("default")
> palette()
[1] "black"  "red"  "green3"  "blue"  "cyan"  "magenta"  "yellow"  "gray"
```

8.6. Graphic export

The simplest way is to use the menu of the Plots tab of RStudio.
Using functions such as `pdf()`, `jpeg()`, `bmp()`, `png()`, `tiff()` (run `?device` to get the whole list) is slightly more technical and will not be addressed here.