1 First R Session

R is a software for statistical analysis and computing. It is used a lot in academic research but it's is also more and more used in companies, industry, banks etc. It is both a programming language and a working environment. R can be used for handling datasets, data visualisation, statistical analysis and the development of statistical algorithms. R is *opensource* and works with UNIX, Windows and Macintosh. It is a free software and you can be downloaded and install it on your personal computer.

This document relies on the books *Le logiciel R* by Lafaye de Micheaux et al. and *Statistiques avec R* by Cornillon *et al.*. Both provide more complete introductions to R.

1.1 Different interfaces

There are several interfaces to use R. First of all, there is the standard interface of R, which contains only very basic features. This minimalisme may be inconvenient for beginners. However, to obtain R you have to start by installing this interface. You obtain it from the plateform CRAN (*Comprehensive R Archive Network*). A much more appealing interface with plenty additional features is the RStudio.

1.1.1 Installation

To install R on your computer follow these instructions. Note that everything is freely available.

1. **Install** R from the plateform CRAN at the adress
   `http://cran.r-project.org`
   Install the latest version.

2. **Install** RStudio from the website
   `https://www.rstudio.com/products/rstudio/download2/`

3. **Install Jupyter Notebook**. There are several ways to install Jupyter. Some instructions are available here:
   `http://jupyter.readthedocs.io/en/latest/install.html`
   The easiest way consists in installing Anaconda from the website
   `https://www.continuum.io/downloads`
   The choice of the Python version (Python 2.7 or Python 3.5) has no importance for us.

4. **Install the R kernel for Jupyter Notebook**. For informations please see here
   `https://github.com/IRkernel/IRkernel/blob/master/README.md`

   For Windows you mainly have to open R and execute the following instructions:

   ```
   install.packages(c('pbdZMQ', 'repr', 'devtools'))
   devtools::install_github('IRkernel/IRkernel')
   IRkernel::installspec()
   ```

   For Mac, it is the same procedure, but you have to start R form the terminal. Just do

```
R
```

in the terminal. Then execute the same commands as for Windows. To shut down `R` do

```
q()
```

Then, you can start Jupyter from the terminal by the command

```
jupyter notebook
```

For Ubuntu, type the following command in the terminal (and use your password)

```
sudo apt-get install libcurl4-openssl-dev libssl-dev libzmq3-dev
```

Then start `R` from the terminal by the instruction

```
R
```

Next do

```
install.packages(c(`crayon`, `pbdZMQ`, `devtools`))
devtools::install_github(paste0(`IRkernel/`, c(`repr`, `IRdisplay`,
  `IRkernel`)))
IRkernel::installspec()
```

Finally quit `R` by the instruction

```
q()
```

Then, you can start Jupyter from the terminal by the command

```
jupyter notebook
```

1.2 My first session with `R`

Start `RStudio` on your computer.

Three windows appear. The most important is the one on the left: the (*console*). The console allows to execute instructions and code. At the end of the displayed text in the console the character `>` invites you to type your first `R` instruction.

```
>
```

This is the *prompt symbol* that tells you that the computer is ready to execute your code.

1.2.1 `R` as a calculator

First of all, you may use `R` as a classical calculator. Let's try to compute $2 + 3$:

```
> 2+3

[1] 5
```

R has understood what we want and displays the right answer. Let's try something slightly more difficult:

```
> 3+8^2*(1-3)

[1] -125
```

We observe that R knows the priority of operations! When R does not understand an instruction, because it is erroneous, an error message is displayed:

```
2+*2

Error: <text>:1:3: '*' inattendu(e)
1: 2+*
^
```

If an instruction is incomplete, R returns the character + instead of the prompt symbol >. You may either complete the instruction or quit the situation by typing Ctrl + c or Echap:

```
> 2*(3+1
+ )

[1] 8
```

There are numerous mathematical functions that are predefined in R. For example:

```
> exp(1)

[1] 2.718282

> sqrt(2)

[1] 1.414214

> abs(-5.7)

[1] 5.7

> round(-5.7)

[1] -6
```

When you perform some forbidden operation, R returns the valeu NaN (for *not a number*) and shows a warning:

```
> sqrt(-2)

Warning in sqrt(-2):  production de NaN

[1] NaN
```

The value `Inf` represent infinity. For example:

```
> 1/0

[1] Inf

> -exp(exp(100))

[1] -Inf
```

You should always keep in mind that numerical calculs is not necessarily exact due to the discretisation of numbers on a machine. Compare the outputs of the following commands:

```
> sin(0)

[1] 0

> sin(pi)

[1] 1.224647e-16

> sin(2*pi)

[1] -2.449294e-16
```

The precision of a computer is of the order of 10^{-16}. This seems very small, put in some cases, theses error may reinforced and lead to dramatic errors. For example:

```
> sin(pi*10^16)

[1] -0.3752129
```

1.2.2 Help in R

R has a very efficient help system. It is quite complete and you may access it by the function `help()` or the command `?` followed by the name of the function on which you are looking for help. For example, you open the help page of the function `abs()` in the following way:

```
> help(abs)
```

or equivalently by

```
> ?abs
```

We observe that both of these commands open the associated help page in the window on the bottom on the right in `RStudio`. It contains a description of the function, the list of arguments and their default value (if defined) and the list of outputs of the function. At the bottom of the page, there may be one or several examples. You may copy-paste them into the console for a better understanding of the usage of the function.

It is a good habit to use the help system extensively. You may progress quickly!

1.2.3 Scripts

Instead of working directly in the command line, you can write a *script*. A script is a text document that contains multiple commands. You can save a script and reuse it in a later session. To develop code, you absolutely have to use scripts.

To create a new script and save it, use the menu. The filename of an `R` script always ends with `.R`

To execute a portion of the commands from an script, select the corresponding lines and hit the bottom Run in the toolbar or do Ctrl + R (Cmd + Enter for Mac). To execute a single line, the one where the curser is blinking, do Ctrl + R. To execute all instruction of the entire script, press the bottom Source.

To insert comments in the script, that is text that should not be executed, write # at the beginning of the comment:

```
> 'ce qui suit' # ne s'affiche pas

[1] "ce qui suit"
```

All text following the symbol # until the end of the line will not be interpreted by `R`.

It is extremely important to comment your code if you wish other people to use it. Or even if you intend to use it again.

1.2.4 Working directory

The *working directory*) is the default directory. This means that this is the directory where your script files are registered or where datasets are expected to be when you upload data to the console.

To know your current working directory write

```
> getwd()

[1] "/Users/tabea/Teaching"
```

To change the working directory, go to *Session* → *Set Working Directory* → *Choose Directory* and select the directory that you need for this session.

It is important to set the working directory at the beginning of every session before doing anything else. Moreover, for a good file management, you do well to create a new directory for every new project that you work on.

1.2.5 Creating R objects

The strength of R (as of any other programming language) consists in the possibility to go much farther than using R just as a simple calculator. We have noticed, that R responds to our commands by displaying the result after the evaluation of an instruction. The result is displayed in the console and then gets lost. This may be okay for simple computations, however it is extremely annoying for more involved problems that cannot be solved by a single instruction. In this case, it is convenient to store the output in some variable. Indeed, to develop programs, that is a set of commands, we have to create variables or objects that save information, can be modified and can be used in other instructions for computing.

Creating R objects is very simple. By assigning a value to a name of variable, the object is created (if it does not yet exist) and it takes the value that we assigned to it. Thus, an assignement evaluates an expression and instead of displaying the result the output is redirected to the variable. Example: We create a variable named x that takes the value 74:

```
> x <- 70 + 4
```

Executing the command does not produce any output, but the object x is created. To display the value of an object, just type its name in the console:

```
> x

[1] 74
```

Now we can use the variable x for further computation:

```
> 3*x

[1] 222
```

To change the value of x, we just assign a new value to x:

```
> x <- x+6
```

Let's check that the value of x has really changed:

```
> x

[1] 80
```

The symbol for assignments is either <- or = Example:

```
> y = 17
> y

[1] 17
```

Concerning the names of objects be creative! Indeed, the names `x, y, z` are very bad examples. It is better to use more informative names. A good choice for the names of variables may render code "readable" and may reduce the amount of comment of a code necessary to understand the program. Let's see with a simple example:

```
> weight <- 52
> height <- 1.65
> BMI <- weight/height^2
> BMI

[1] 19.10009
```

In `R` variable names must start with a letter. It may contain numbers or the characters `.` and `_`

1.2.6 Deleting objects

The *work space* is some kind of space where all `R` objects are stored that have been created during a session. That is, it is the list of objects currently known to `R`. To see all objects that have been created since the beginning of the session, write

```
> ls()

[1] "BMI"    "height"  "weight" "x"      "y"
```

or use the window on the upper right.
To delete for instance the variable `x` from the work space, type

```
> rm(x)
```

and check that `x` has indeed disappeared:

```
> ls()

[1] "BMI"    "height"  "weight"     "y"
```

To clean the whole work space, that is to delete all objects at once, we write

```
> rm(list=ls())
> ls()

character(0)
```

1.2.7 Leaving RStudio

When closing RStudio, you are asked whether you want to save an image of the current work space, that is all objects from the work space from the current session, so that they can be uploaded in a later session.

2 Vectors

We have seen how to define a variable to store a single numerical value. However, in statistics, in general we need *vectors* of numerical values. In R a vector is created by the function c( ) (c for *concatenate*):

```
vec1 <- c(5,8,3)
vec1

[1] 5 8 3
```

You can also concatenate vectors:

```
vec2 <- c(vec1,0,vec1)
vec2

[1] 5 8 3 0 5 8 3
```

Here some examples for computing with vectors:

```
vec1+2

[1]  7 10  5
```

```
5*vec1

[1] 25 40 15
```

```
vec1+vec1

[1] 10 16  6
```

```
vec1*vec1

[1] 25 64  9
```

```
vec1^2

[1] 25 64  9
```

You can see that it's quite simple and intuitive. However, you always have to be careful with vector dimensions:

```
vec1+vec2

Warning in vec1 + vec2:  la taille d'un objet plus long n'est pas multiple de
la taille d'un objet plus court

[1] 10 16  6  5 13 11  8
```

We observe that despite the error message (read them!!), `R` returns a result. Here, `R` repeats the vector `vec1` two times to create a vector of the same size as vector `vec2`. In `R` it is a common method when dimensions do not correspond, the smaller object is repeated as often as necessary to obtain some computation that makes sense. (However, make sure that this corresponds exactly to what you wish to do!).

2.1 Mathematical functions for vectors

In `R` there are a huge amount of very useful, predefined mathematical functions. Here's a list of functions that apply to vectors:

- `sum()`
- `prod()`
- `length()`
- `min()`
- `max()`
- `which.min()`
- `which.max()`
- `sort()`
- `order()`
- `mean()`
- `var()`

- `sd()`
- `median()`

Exercice 1. Find out what the above functions do. More precisely, start `RStudio`, define some toy vectors, apply the above functions and try to understand the output. If you do not understand a function, use the help function `help()` or the command `?` followed by the function's name.

2.2 Generate pseudo-random variables

In `R` there are functions to generate realisations from random variables with a given probability distribution. For instance, the following command generates 10 realisations of the standard uniform distribution $U[0,1]$:

```
runif(10)

 [1] 0.28882472 0.06704419 0.72498083 0.74772676 0.93207037 0.84667471
 [7] 0.36957618 0.26681129 0.94650848 0.05227033
```

Execute the command for a second to observe that the output values change:

```
runif(10)

 [1] 0.40777739 0.44542213 0.02337936 0.81132470 0.70175204 0.02875725
 [7] 0.97466179 0.20489910 0.12958125 0.94587459
```

Indeed, the function `runif( )` is a generator of *pseudo-random variables*. We say *pseudo*, since they seem to be realisations of a veritable random variable. (The construction of pseudo-random variables is a whole research field that is the development of algorithms that imitate randomness.)

Likewise, there is a generator of (pseudo-)realisations of the standard normal distribution:

```
rnorm(5)

[1]  0.83728215 -0.21270838  0.47703402  0.08815319  0.48344911
```

You may have noticed that the argument of the functions `runif` and `rnorm` is the number of realisations (also called the sample size).

To generate realisations of the normal distribution $\mathcal{N}(\mu, \sigma^2)$ with mean μ and variance σ^2, we still use the function `rnorm( )` with two additional arguments to indicate the parameter values of the normal distribution. Notice that in `R` these are the mean μ and the *standard deviation* σ (instead of the variance σ^2). Hence, two obtain 20 realisations of the normal distribution $\mathcal{N}(-5, 2)$ we use the instruction:

```
n <- 20
mu <- -5
sig <- sqrt(2)
rnorm(n,mu,sig)

 [1] -6.318289 -4.270383 -7.167644 -6.776132 -4.970636 -5.121694 -5.722180
 [8] -6.357610 -7.339206 -4.405400 -5.091744 -4.468014 -5.383254 -6.367177
[15] -3.285406 -5.705408 -5.304806 -5.521791 -4.868823 -6.676552
```

Exercice 2. Starting from the next exercise on, we will write all code in `R` scripts. To do this in a proper way, let's do some preparatory work.

- First of all, create a new directory (named *StatisticsLecture*, for instance) where you will save all files associated with this lecture.
- Next, choose this new directory to be your working directory for this session.
- Then open a new script file: go to *File* → *New File* → *R Script*
- Start your script with a comment of the type
  `# Working Sheet: My first R session, date`
- Finally, save your script file to the directory *StatisticsLecture*. Choose an informative filename, something like *FirstRSessionExercises*.

Exercice 3. The sample mean is an estimator of the theoretical mean of a distribution. Let's check empirically the quality of this estimator by using simulations.

- Consider the normal distribution $\mathcal{N}(10, 1)$. Generate a sample of size 20 and compute the associated sample mean. Compare the result to the theoretical mean. Repeat the operations several times in order to check the stability of the value of the sample mean over different datasets.
- Perform the same analysis for the normal distribution $\mathcal{N}(10, 10)$ and sample size 20. What do you observe?
- The normal distribution is a symmetric distribution. Hence, its theoretical skewness equals 0. Check whether the empirical skewness $\alpha_{\mathbf{x}}$ is close to 0 for datasets from the normal distribution. Try out normal distributions with different parameter values and different sample sizes. Recall the definition of the empirical skewness $\alpha_{\mathbf{x}}$ associated with the sample $(x_1, \ldots, x_n)$:

$$\alpha_{\mathbf{x}} = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x}_n)^3}{s_{\mathbf{x}}^3}, \qquad \text{où} \qquad s_{\mathbf{x}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x}_n)^2}.$$

- Same question for the empirical kurtosis $\beta_{\mathbf{x}}$ of a dataset from the normal distribution that is supposed to be close to 0. Recall the definition:

$$\beta_{\mathbf{x}} = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x}_n)^4}{s_{\mathbf{x}}^4} - 3.$$

- Add comments to your script with your observations and conclusions.

Exercice 4. Let $X_1, X_2, \dots$ be independent random variable with standard normal distribution. Set $Y_k = \sum_{i=1}^{k} X_i^2$. It is well known that the distribution of Y_k is the chi-square distribution χ^2_k with k degrees of freedom.
According to the central limit theorem, we have

$$\sqrt{\frac{k}{2}}\left(\frac{1}{k}\sum_{i=1}^{k} X_i^2 - 1\right) = \frac{1}{\sqrt{2k}}(Y_k - k) \xrightarrow{\mathcal{L}} \mathcal{N}(0,1), \qquad k \to \infty.$$

This implies that the distribution of Y_k is approximately the normal distribution $\mathcal{N}(k, 2k)$ for k sufficiently large:

$$\chi^2_k \approx \mathcal{N}(k, 2k).$$

The aim of the exercise consists in an empirical study to determine the order of k for which the approximation is good. More precisely, we will compare summary statistics of the distribution of Y_k to the corresponding theoretical values of the normal distribution $\mathcal{N}(k, 2k)$.
In `R` to generate n realisations of the chi-square distribution χ^2_d with d degrees of freedom we use the command `rchisq(n,d)`.

- For every k in $\{1, 10, 100, 1000\}$ generate a dataset of size 1000 from the χ^2_k distribution, and compare the associated sample mean to the mean of the normal distribution $\mathcal{N}(k, 2k)$. What do you observe?
- Same question for the sample variance $s^2_{\mathbf{x}}$: For every k in $\{1, 10, 100, 1000\}$ generate a dataset of size 1000 from the χ^2_k distribution, and compare the sample variance to the theoretical variance of the normal distribution $\mathcal{N}(k, 2k)$. Comment the results.
- Same question for the skewness.
- Same question for the kurtosis.
- Include comments in your script with answers to the above questions and detailed conclusions.

3 Different data types

`R` is not limited to variables of the numerical type. The instruction `typeof(x)` returns the type of the object `x`. Let's enumerate the different data types.

3.1 Numerical type (*numeric*)

Among objects of numerical type, we distinguish integer values (*integer*) and real numbers (*real* or *double*). Let's define two variables and check their type:

```
var.a <- 1.7
typeof(var.a)
```

```
[1] "double"
```

```
var.b <- -3
typeof(var.b)

[1] "double"
```

The variables `var.a` and `var.b` are both of type *double*. To obtain a variable of *integer* type, we apply the function `as.integer`. The interest of the *integer* type is that it is more economic with respect to memory for storing the value.

```
var.c <- as.integer(var.b)
typeof(var.c)

[1] "integer"
```

3.2 Boolean or logical type (*logical*)

`R` is able to give answers to simple questions. Not in terms of "yes" and "no", but in terms of `TRUE` and `FALSE`. Here are some examples:

```
# equality of values of  var.a and var.b ?
var.a==var.b

[1] FALSE
```

```
# var.a is larger than var.b ?
var.a>var.b

[1] TRUE
```

Theses responses are of the boolean type. For the boolean type no other values than `TRUE` and `FALSE` are possible (there is no value for answers of the type "I don't know" or "I don't care about"). The "questions" in the above examples are called *logical conditions*, since the result of logical type.

Also the output of the functions that test the type of variable `x` (starting by `is.`) is boolean:

```
is.numeric(var.b)

[1] TRUE
```

```
is.integer(var.b)

[1] FALSE
```

```
var.x <- FALSE
is.logical(var.x)

[1] TRUE
```

`TRUE` and `FALSE` can be written more briefly by `T` et `F`.

When necessary, boolean variables are converted: `TRUE` to 1 and `FALSE` to 0. This is illustrated in the following example:

```
TRUE + FALSE

[1] 1
```

```
TRUE/2 - var.b

[1] 3.5
```

```
(TRUE + T)^2 + FALSE*F + T*FALSE + F

[1] 4
```

3.3 Character string (*character*)

Character string are created by using ' or ").

```
string1 <- "I love maths"
string1

[1] "I love maths"
```

```
typeof(string1)

[1] "character"
```

```
string2 <- 'Moi aussi !'
string2

[1] "Moi aussi !"
```

```
is.character(string2)

[1] TRUE
```

3.4 Type conversion

Type conversions are possible thanks to functions whose names starts by `as.`

However, you may be very careful about the meaning of these conversions. The `as.`-functions always return a result even if its complete nonsense. See the following examples:

```
# Conversion to a character string
as.character(2.3)

[1] "2.3"
```

```
# Conversion of a character string
string3 <- "2.3"
as.numeric(string3)

[1] 2.3
```

```
# Conversion of a character string
as.integer(string3)

[1] 2
```

```
# Conversion is impossible
as.integer(string1)

Warning:  NAs introduits lors de la conversion automatique

[1] NA
```

4 Data structures

`R` offres the possibility to organise collections of objects of the same or of different type. In statistics, vectors and dataframes are commonly used. Lists will be treated later.

4.1 Vectors

This data structure is the most simple. It is a *sequence of data of the same type*. We have already seen vectors of numerical type, but there are also vectors of booleans or of character strings.:

```
vec.num <- c(1,4,9,2,0)
vec.num

[1] 1 4 9 2 0
```

```
vec.char <- c("R","c'est","trop","facile","!")
vec.char

[1] "R"      "c'est"  "trop"   "facile" "!"
```

A vector of booleans may be obtained by applying a logical condition on vectors. Example:

```
vec.logique <- (vec.num>=4)
vec.logique

[1] FALSE  TRUE  TRUE FALSE FALSE
```

In this example, the elements of the vector `vec.num` are compared one by one to the value 4 and the output is written to the vector `vec.logique`.

In the next example, we perform a comparison element by element of two vectors:

```
vec.char2 <- c("R","c'est pas","mon","truc","!")
vec.char2

[1] "R"         "c'est pas" "mon"       "truc"      "!"

vec.logique2 <- (vec.char == vec.char2)
vec.logique2

[1]  TRUE FALSE FALSE FALSE  TRUE
```

4.1.1 Automatic conversion

By definition, all elements of a vector are of the same type. When mixing different types during the creation of a vector, `R` automatically converts the type to the most general type, as can be seen from the following example:

```
c(3,TRUE,7)

[1] 3 1 7
```

```
c(3,T,"7")

[1] "3"    "TRUE" "7"
```

4.1.2 Regular sequences

To create vectors, on may use regular sequences.

Exercice 5. Execute the following commands and use the help to understand their meaning.

- l'utilisation des deux-points `:`,
- la fonction `seq()` et ses arguments `length` et `by`,
- la fonction `rep()` et ses arguments `times` et `each`.

```
2:5
```

```
.5:4
```

```
12:-4
```

```
seq(2,6)
```

```
seq(2,6,by=.2)
```

```
seq(2,6,length=4)
```

```
rep(1,5)
```

```
rep(1:3,times=4)
```

```
rep(1:3,each=4)
```