R Tutorial

Fabien Baeriswyl, Jérôme Reboulleau

R Tutorial

This tutorial is designed to review the basic tools that `R` has to offer in order to undertake statistical tasks.

Downloading R and RStudio

First, you have to download `R`:

1) **With Windows:** Go to

   https://cran.r-project.org/bin/windows/base/

   and use the first link on the page.

2) **With OS:** Visit

   https://cran.r-project.org/bin/macosx/

   and download the .pkg extension file.

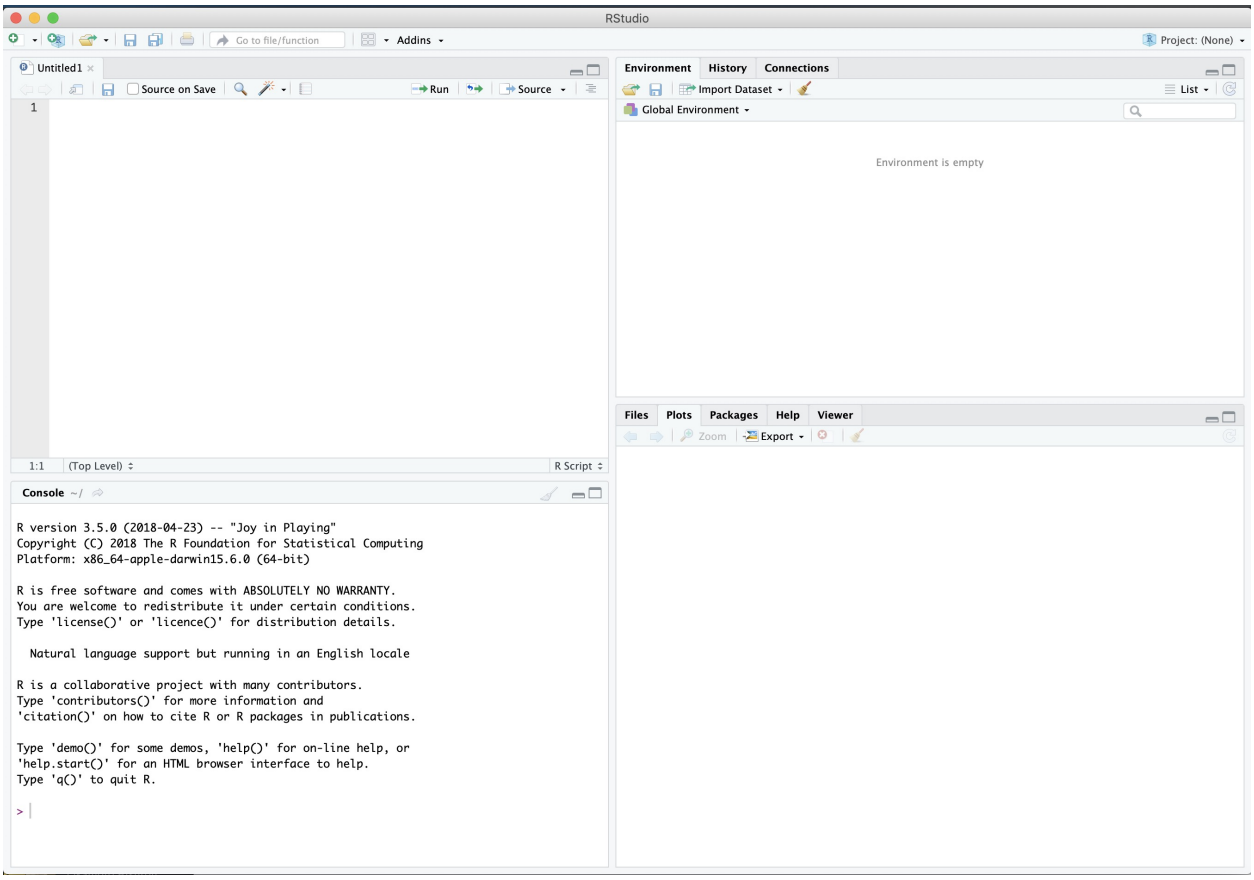
3) **With Linux:** you should find it on your list of packages.

A very useful interface for `R` is `R Studio`, which can be downloaded from the following link:

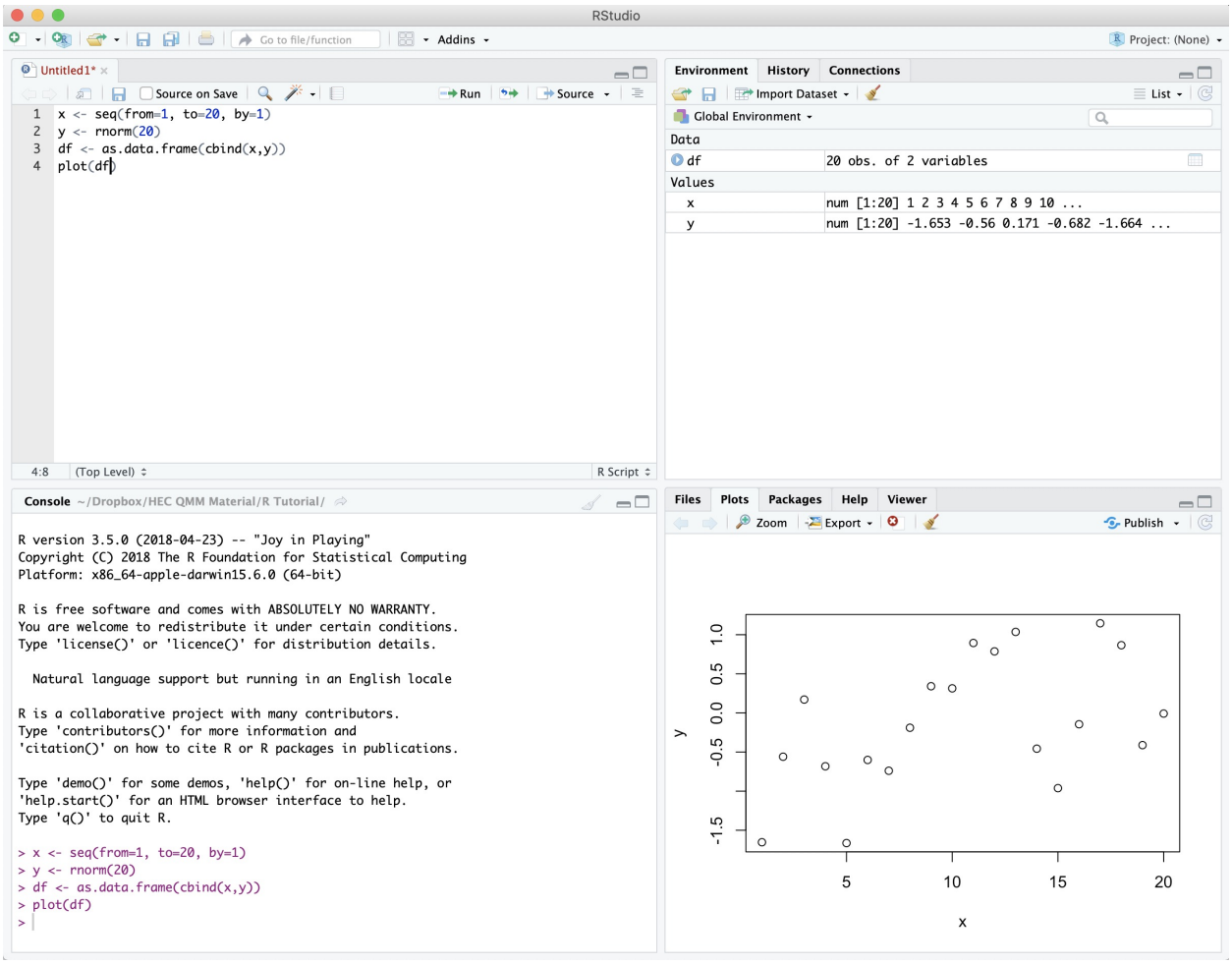
https://rstudio.com/products/rstudio/download/.

Simply download the `RStudio Desktop` (DO NOT PAY ANYTHING FOR IT, THE FREE VERSION IS MORE THAN ENOUGH!).

`R Studio` offers a nice and easy interface to use the tools that we will need in this course. After downloading it and opening it, go to File -> New File -> R Script to create a new script. This will allow you to save your code in a designed space. Your screen should look like the following:



The upper-left panel is the script per se. It is where you can write your code and save it in a specific format to reopen it after. The bottom-left panel is the console, where you can execute single (or multiple) line(s) of code, but this won't be saved (and should not) since this is a dedicated space to execute simple tasks related to the bigger work you are undertaking. When you execute (or "Run"") the lines of code in the script, it will automatically appear in the console, since this is a kind of history for the commands you run. The next screenshot illustrates the above:



One can see that in the upper-left panel, there is some code that we will cover in details later. Once executed, this code appears in the bottom-left panel, in the consol, in purple. The `x` and `y` vectors also appear as "Values" in the Environment in the upper-right panel, as well as the `df` data frame, which consists of 20 observations of 2 variables. Also, in the bottom-right panel, one can see the result of the third line of code, which is a plot.

The upper-right panel is the environnement in which you will see your vaiables, dataframes,... The other tabs, History and Connections, will not be used all that much. Connections is where you can link your work to a Git version control. You will see this in Data Science and Programming Tools, if you take these lectures. The lower-right panel finally is where you find your Files (i.e. the files in your working directory), your plots, your packages loadout and the help pages. We will cover these functionalities throughout this tutorial.

Before getting started, we want to emphasise that the only way for your to learn `R` is to practice by yourself. Do not only look at the solutions of the assigments, but try them on. Try to understand the logic, the structure of the code since this should not be too complicated, especially if you have already learn how to code in other languages. `R` provides more of a description language, in which you do not have to worry about variables declaration and such, as you would with a programming language such as C++.

Note that, throughout this semester, I will refer in the solutions to `R` version `4.0.1` and `RStudio` version `1.3.959` (I am using an OS system). Some versions might have some (minor) differences and it might be good to use the same versions in order to have the same results. In particular, some function nomenclatures can change over time. So if you have a doubt, these are my settings.

Good practice when coding

If you are new to coding, you might want to follow some of the principles that are useful to know when starting your own tasks:

1) You should always work in separate folders for a particular Assignment/Homework/Project. We recommend to create a folder, e.g. Assignment1 and to then split this folder in a Data folder, Code folder, etc. for clarity.

2) You should have two versions of your code: one that is clean, working and compiling (no error stopping the code from working) and one that is your draft version. This allows you to work freely on one version while saving a correct version on the side, with safe-tested elements and lines of code. This also prevents you from losing the important part of your work if anything happens.

3) You should always save and store backup versions.

4) You should comment your code to remember the meaning or the purpose of some chunks of code. To do so, add a "#" and put your comments after. Comments will appear in green on your code. You will see it a bit later in this tutorial.

5) When you do not know how to do something in `R`, google it! This will save numerous e-mails to the professor or the assistants and, more often than not, the answer you are looking for is already somewhere online. And do not worry, we google `R` related queries all the time. So this is a sign that you know what you are doing. The more useful ressources online for this matter are R-bloggers, available at

   https://www.r-bloggers.com/

   and CrossValidated, available at

   https://stats.stackexchange.com/.

   Often, these websites also offer tutorials on how to use `R` or specific `R packages`.

For more best practice advice, please visit the following link:

https://code.tutsplus.com/tutorials/top-15-best-practices-for-writing-super-readable-code--net-8118.

Getting started: R as a powerful calculator

First, note that `R` can be used as a kind powerful calculator, although it should not be limited to that. For example, typical computations can be undertaken by writing and executing the following in your script (the result will be displayed in the console):

```
5+10
```

```
## [1] 15
```

```
2-3
```

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

```
8*exp(10)/log(2)
```

```
## [1] 254219.8
```

You already see that you can also use functions, such as the exponential or the logarithm (base e with the `log()` function) to undertake your computations. Many mathematical functions are coded in the base package loaded with `R`: for the more complex and less frequent functions, one can use various `packages` to directly undertake some operations that would take multiple lines of code. These packages are also useful since they are also often made computationally efficient by their author(s). But more on `R packages` a bit later.

Vectors, matrices and various objects

In R, it is usual, as with other programming tools, to store information in specific objects, such as vectors or matrices. The information stored in vectors can be of any type: character (e.g. names), double (decimal numbers), integer, a combination of all these...

To create an empty vector, one uses:

```
vec <- c()
```

The "c" stands for concatenation. Note that the "<-" can be replaced by "=" but to avoid confusion, one uses the most common assignation operator in R, i.e. "<-". Remember that assignation works from right to left: one assignates the value on the right to the value on the left. In the example above, one assignate a NULL (empty) vector to the variable "vec" which now appears in your Global Environment in your upper-right panel.

One can also directly create a vector with information stored in it :

```
vec2 <- c(1,2,3)
colours <- c("blue", "red", "yellow")
```

We have now assigned the values 1, 2 and 3 to `vec2` and the values (or nouns) blue, red and yellow to the `colours` vector. One can check the types of these vectors by using the `typeof()` command and their length by using the `length()` command:

```
typeof(vec2)
```

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

```
typeof(colours)
```

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

```
length(vec2)
```

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

```
length(colours)
```

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

The first vector is of type "double", with decimal numbers in it (even though they're integer) and the second of type "character", with names, signalled by the fact that we entered these values with "". Note that the following vector:

```
vec4 <- c("1", "2", "3")
typeof(vec4)
```

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

is also of type "character", since we entered the values in it with "".

To now access specific elements of the vector, one uses the nomenclature vector[]. The numerotation starts at 1 (with Python it is 0... which can be confusing) and so, if one wants to access the first value of the `colours` vector, one types:

```
colours[1]
```

```
## [1] "blue"
```

which returns "blue", as expected.

A vector, as a mathematical object, has one dimension. To go beyond this paradigm, one uses matrices (in 2 dimensions) or arrays (in higher dimensions).

To create an empty 2x2 matrix, made up of "NA" elements, use:

```
mat <- matrix(NA, ncol=2, nrow=2)
```

where one specifies the number of columns (ncol) and the number of rows (nrow). One can then access elements in the matrix by the nomenclature matrix[r,c] to access the rth row and the cth column. If we create a matrix with elements 1,2,3,4:

```
(mat2 <- matrix(c(1,2,3,4), ncol=2, nrow=2))
```

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

and if we want to access "3" in that matrix, which is in row 1 and column 2, one will therefore write:

```
mat2[1,2]
```

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

To replace an element, for example to replace the "1" by "0", one would simply change that specific entry by assigning a new value:

```
mat2[1,1] <- 0
mat2
```

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

Note that the length of a matrix is its number of elements, in that case 4. To get its dimension (it is a 2x2 matrix), use `dim()`:

```
dim(mat2)
```

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

Another very useful type of object is the data frame. For example, if one wants to create a data frame from the `vec2` and `colours` vectors, one uses the following code:

```
df <- as.data.frame(cbind(vec2, colours))
print(df)
```

```
##   vec2 colours
## 1    1    blue
## 2    2     red
## 3    3  yellow
```

In the above code, note the following: we used the `cbind()` function to bind the vectors into two distinct columns (cbind means column bind - a similar version exists row-wise, using `rbind()`). We then coerced this new object, made of two binded vectors, to be of a data frame type by using `as.data.frame()` (read the code as a convolution, start your operation inside and go outside). To display the rth row of this data frame, use the nomenclature dataframe[r,] (letting a blank for the specification of the column you want to access in the format we covered before, so that it selects all columns); to display the cth column, use dataframe[,c]. Here, we display the first row first and the second column then:

```
df[1,]
```

```
##   vec2 colours
## 1    1    blue
```

```
df[,2]
```

```
## [1] "blue"   "red"    "yellow"
```

Note that the vectors in the data frame still have their respective name, i.e. vec2 and colours. They can therefore also be accessed by using:

```
df[,"vec2"]
```

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

```
df[,"colours"]
```

```
## [1] "blue"   "red"    "yellow"
```

You can also rename them by using the `colnames()` function

```
colnames(df) <- c("col1", "col2")
print(df)
```

```
##   col1   col2
## 1    1   blue
## 2    2    red
## 3    3 yellow
```

You can now also access the first column of your dataframe by the nomenclaure dataframe$col1, i.e.

```
df$col1
```

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

Uploading data

We will now see how to dowload data from external sources. To be able to download data, you must have saved your script and be working in the exact same "working directory" of the data file. You can also have your data somewhere else but in that case, you have to specify where you want `RStudio` to look for it. To know in which working directory you are, use the `getwd()` function. To set it manually, use the `setwd()` command, specifying the exact path where your data are. However, if you don't know how to look for it, `RStudio` allows you to set it by using the tab Session -> Set Working Directory -> Choose Directory... Then simply go to the folder where your data file is and without selecting the file per se but only the folder in which it is, click "Open".

Note that if you want to set a specific working directly in `setwd()`, the nomenclature is as follow, as stressed in http://www.sthda.com/english/wiki/running-rstudio-and-setting-up-your-working-directory-easy-r-programming#change-your-working-directory:

1) For Windows: Use the command `setwd("c:/Documents/my/working/directory")`, changing for the path where your document is;
2) For MacOS: Use the command `setwd("/path/to/my/directory")`, changing for the path where you document is.

Once you locate your data file, you are ready to load it without having to worry about file extension.

The most common format you will encounter in this class is the `.csv` extension. To load data from such a format, use the `read.csv()` command (or `read.table()`). Note that, if this is a `.txt` that you have to load, use `read.delim()`. In this case, we want to read the `dataex.csv` file, found on Moodle.

```
dataframe <- read.csv("dataex.csv")
```

We see that dataframe now appears in the Global Environment with an arrow on its left. Clicking the arrow shows the column names alongside their types (numeric here) and the first few values. It also summarises that the data frame is made of 50 observations of 3 variables (i.e. 3 columns).

Suppose you want to add to this dataframe a column made of 1s only and a sequence column, ranging from 1 to 50. Then, you can just create these sequences (same size as the original dataframe, so 50 observations) by using the `rep()` (for replicate, it replicates a certain value n times) and `seq()` (it creates a sequence starting somewhere, ending somewhere else with a possibility to gve the increment size) and "cbind" them to the original data frame, or create a new data frame with it, say dataframe2. You might then want to save it and for that, you want to use the `write.csv()` command.

```
seq1 <- rep(1, 50) #a sequence of 1 replicated 50 times
seq2 <- seq(from=1, to=50, by=1) #a sequence from 1 to 50 by 1, can also be written seq(1,50,1)
dataframe2 <- as.data.frame(cbind(dataframe, seq1, seq2))
write.csv(dataframe2, "dataframe2.csv", row.names=FALSE)
```

Note that, above, we used the row.names=FALSE to prevent the CSV file to create an index column, equivalent in that case to `seq2`.

Basic statistical tools

We may want to obtain information about some statistical quantities of the columns making up the `dataframe` data frame. The basic tools that you might already know are the following: the `mean()` (for the sample mean), `median()` (for the sample median), `sd()` (for the sample standard deviation), `quantile()` (for the sample quantiles), `cov()` (for the sample covariance) and `cor()` (for the sample correlation) commands.

You can find for example the mean, the median and the standard deviation of column x of `dataframe()` by typing:

```
mean(dataframe$x)
```

```
## [1] -0.1558902
```

```
median(dataframe$x)
```

```
## [1] -0.1559878
```

```
sd(dataframe$x)
```

```
## [1] 0.9037593
```

For the `quantile()` function, one can save the output of the function in an object named, for example quant, and access a specific quantile, say the 50% quantile by using the nomenclature quantile(x)[3], or if the object is directly named as follow:

```
quant <- quantile(dataframe$x)
quant[3]
```

```
##        50%
## -0.1559878
```

We see that typing quant[3] actually gives us the information about it being the 50% quantile. If we want to only keep track of the number (sometimes you have to do it to be able to use it explicitely), you can coerce your answer to be numeric, using:

```
as.numeric(quant[3])
```

```
## [1] -0.1559878
```

To use the covariance `cov()` or the correlation `cor()` commands, one needs two inputs, for example for the covariance and the correlation between columns x and z in `dataframe`:

```
cov(dataframe$x, dataframe$z)
```

```
## [1] 0.1918525
```

```
cor(dataframe$x, dataframe$z)
```

```
## [1] 0.6307821
```

Using a matrix format, one can obtain the correlation matrix directly as follow:

```
cor(as.matrix(dataframe))
```

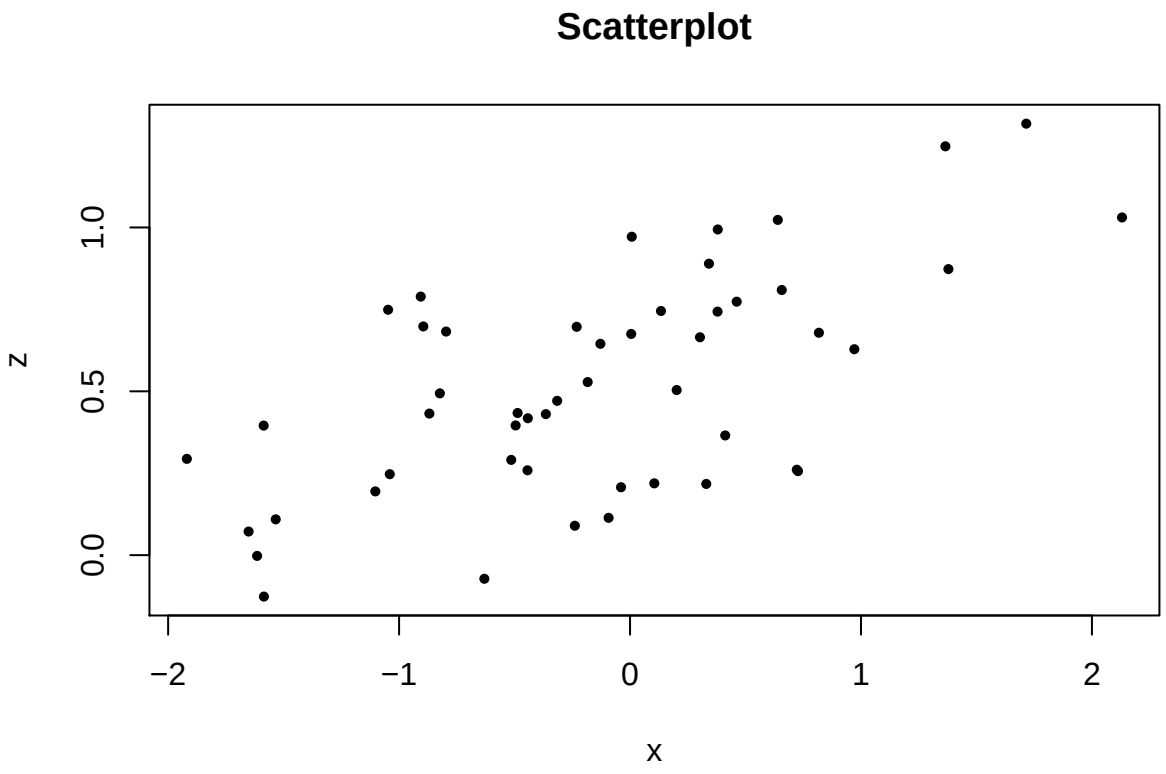
```
##           x          y         z
## x 1.0000000 0.14033876 0.63078209
## y 0.1403388 1.00000000 0.09242662
## z 0.6307821 0.09242662 1.00000000
```

This matrix shows the correlation between all variables considered. Note that this matrix is symmetric.

More advanced statistical functions and plots

In the course, you reviewed the basic of simple linear regression. To undertake a linear regression in `R`, you might first need an idea about the relationship between two variables of interest. We saw above that x and z seem to be correlated, so we might want to assess this with a plot.

```
plot(dataframe$x, dataframe$z, main="Scatterplot", xlab="x", ylab="z", cex=0.7, pch=16)
```



In the above, note that we customised the basic `plot` function, but it works as follow: first, in order, give the x-axis and the y-axis; then, in no particular order, add the main title, the label for the x and the y axes, the

size of the points (cex) and their type (pch). You can find online lists of customisation and parameters or directly access the help function to find out about the above function by typing:

```
?plot
```

```
## Help on topic 'plot' was found in the following packages:
##
##   Package               Library
##   graphics              /Library/Frameworks/R.framework/Versions/4.0/Resources/library
##   base                  /Library/Frameworks/R.framework/Resources/library
##
##
## Using the first match ...
```

A single ? in front of a command will lead to its help page if it is known by your current (and loaded) version of `R`. In case you use functions coming from a specific package (more on that later), you will need to use the `??command` to access an help page dedicated to find matching functions to your needs in the case you have not loaded that package yet in your `R` session.

Now, you may want to fit a linear model of the form $x = \beta_0 + \beta_1 z$ since the two seem correlated. You will do so by using the `lm()` command and to get the coefficients and the useful about the regression, just use `summary(yourmodel)` where your model is your linear fit name:

```
mod <- lm(dataframe$x~dataframe$z)
summary(mod)
```

```
##
## Call:
## lm(formula = dataframe$x ~ dataframe$z)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.38593 -0.42153 -0.07279  0.52044  1.41531
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -1.0307     0.1848  -5.576 1.11e-06 ***
## dataframe$z   1.6939     0.3008   5.632 9.10e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.7085 on 48 degrees of freedom
## Multiple R-squared:  0.3979, Adjusted R-squared:  0.3853
## F-statistic: 31.72 on 1 and 48 DF,  p-value: 9.103e-07
```

One accesses multiple information: the estimated regression coefficients, their respective standard errors, the t-value associated to a t-test to check whether the coefficients are statistically different from 0 or not and the associated p-values. The stars are reflection of the level of significance. Here, it seems that both the slope and the intercept are statistically different from 0. The R^2 and adjusted R^2 are also reported. As well as an F-test (but that's for later).

In your Global Environment, you now see your linear model as a list of various elements. You can retrieved, among others, the coefficients, the residuals of your models, the fitted values,... For example, to retrieve the coefficients of the regression (also displayed in the summary above), you can type, using your fitted model name:

```
mod$coefficients
```

```
## (Intercept) dataframe$z
##   -1.030684    1.693935
```

Note also that the `attach()` function allows you to access x,y and z from dataframe directly if you attach it:

```
attach(dataframe)
head(x) #head() displays the first 6 observations of a specific vector, data frame...
```

```
## [1]  0.461825862 -0.906283241  2.130253443  0.007487846 -0.895132841
## [6] -0.128444132
```

Note that you only use `attach()` once: if you do it twice, then it will mask x and you will not find it anymore. Similarly, when you are done with accessing the variables from this dataframe by their column names, you should always use the `detach()` command:

```
detach(dataframe)
```

The object x cannot be found anymore (you will get an error if you try - please try).

Downloading and loading packages

Often, it is useful to use dedicated packages to undertake your statistical tasks in `R`. For example, for more advanced functions, for procedures that simplify handling of complex data and so on.

A very useful package is the `dplyr`, for which you can find a complete introduction to on this link: https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html. This package allows you to efficiently treat data and to perform specific tasks with data frames, objects called `tibbles` in the nomenclature of this package.

When you want to use a package, you need to first install it in your `R` version. To do so, use the `install.packages("nameofthepackage")` command. In this case, use:

```
install.packages("dplyr")
```

Once you have downloaded it, the files and functions are ready in your `R`. But each time you want to use this package, that is in each of your `R` session, you have to load it, by using the `library(nameofthepackage)` command. In this case:

```
library(dplyr)
```

For example, the `filter()` function in the `dplyr` package is a very useful verb (the functions in the `dplyr` packages are called this way) to filter some rows of a dataframe.

Suppose we have a new column in our `dataframe` object, consisting of 20 1s and 30 2s corresponding to two groups of observations:

```
seq3 <- c(rep(1,20), rep(2,30))
dataframe <- cbind(dataframe, seq3)
```

We want to filter the rows with seq3=1. To do so, we could use:

```
dataframe[dataframe[,"seq3"]==1,]
```

```
##               x          y           z seq3
## 1   0.461825862 2.88707771  0.77349186    1
## 2  -0.906283241 4.43729005  0.78903855    1
## 3   2.130253443 1.69284155  1.03051475    1
## 4   0.007487846 0.71109843  0.97180330    1
## 5  -0.895132841 0.10111119  0.69787621    1
```

```
## 6  -0.128444132 1.20969874  0.64502630    1
## 7  -1.047574796 0.34479392  0.74884933    1
## 8  -0.239086116 0.77465974  0.08968529    1
## 9  -1.040253002 0.47981271  0.24708738    1
## 10 -0.443975356 0.29675847  0.25887485    1
## 11  0.378885455 0.05732184  0.74319498    1
## 12 -0.823506671 0.77523619  0.49372266    1
## 13 -0.442491254 0.10700244  0.41784267    1
## 14 -1.614815319 2.31637628 -0.00236244    1
## 15  0.412143259 0.46584620  0.36530579    1
## 16 -0.486807527 0.77198674  0.43377770    1
## 17 -1.534231816 0.75815458  0.10925320    1
## 18  1.378537951 1.32833785  0.87295549    1
## 19 -1.918975195 0.67683921  0.29377518    1
## 20 -0.796430364 0.27361468  0.68236446    1
```

Which is to say that we select the rows of `dataframe` for which the seq3 column is 1 (note the double ==, this is to check a logical condition, since the single = is reserved for assignation).

But with `dplyr` verbs, it suffices to write:

```
filter(dataframe, seq3==1)
```

```
##               x          y           z seq3
## 1   0.461825862 2.88707771  0.77349186    1
## 2  -0.906283241 4.43729005  0.78903855    1
## 3   2.130253443 1.69284155  1.03051475    1
## 4   0.007487846 0.71109843  0.97180330    1
## 5  -0.895132841 0.10111119  0.69787621    1
## 6  -0.128444132 1.20969874  0.64502630    1
## 7  -1.047574796 0.34479392  0.74884933    1
## 8  -0.239086116 0.77465974  0.08968529    1
## 9  -1.040253002 0.47981271  0.24708738    1
## 10 -0.443975356 0.29675847  0.25887485    1
## 11  0.378885455 0.05732184  0.74319498    1
## 12 -0.823506671 0.77523619  0.49372266    1
## 13 -0.442491254 0.10700244  0.41784267    1
## 14 -1.614815319 2.31637628 -0.00236244    1
## 15  0.412143259 0.46584620  0.36530579    1
## 16 -0.486807527 0.77198674  0.43377770    1
## 17 -1.534231816 0.75815458  0.10925320    1
## 18  1.378537951 1.32833785  0.87295549    1
## 19 -1.918975195 0.67683921  0.29377518    1
## 20 -0.796430364 0.27361468  0.68236446    1
```

which is more readable and accessible, especially when you are unfamiliar with `R`.

Theoretical questions in reviewing statistical knowledge and R

The following questions are aimed at reviewing some statistical knowledge on simple linear regression and to then see how to use `R` effectively to illustrate these concepts.

Question a)

You saw that the (sample) Pearson coefficient r is given by the formula

$$r = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sqrt{(\sum(x - \bar{x})^2)(\sum(y - \bar{y})^2)}}.$$

What does it measure? In particular, what kind of correlation? What are the potential drawbacks of such a correlation coefficient?

Pearson correlation coefficient measures the linear association between two random variables X and Y of interest. The formula you were given in this question is the sample analogue to the population Pearson correlation coefficient ρ defined as

$$\rho(X, Y) = \frac{cov(X, Y)}{\sigma_X \sigma_Y}$$

where $cov(\cdot, \cdot)$ is the covariance function, σ_X is the standard deviation of X and σ_Y is the standard deviation of Y, respectively. A major drawback of this measure is that it is limited to the linear paradigm and therefore does not effectively measure the correlation when the random processes at hand are related quadratically, for example. To circumvent such problems, one can have a look at Spearman's rho or Kendall's tau, for example. But this is beyond the scope of this course. This is for you to recognise that Pearson correlation coefficient is great in a linear setting but is not as powerful in other situations.

To measure Pearson's sample correlation coefficient, we saw that you need to use the `cor()` function in `R`. For example, in `dataframe`, the correlation between x and y is given by:

```
cor(dataframe$x, dataframe$y)
```

```
## [1] 0.1403388
```

Question b)

When one is conducting hypothesis testing, why does one compare the test statistic to a specific critical value? In other words, why does one choose a specific (distribution) law to which one compares the test statistic? (*Hint: Think of hypothesis testing on $\bar{X}$, for example, and remember your first statistics course.*)

In hypothesis testing, we always make a distributional assumption for our test statistic under the null hypothesis H_0. For example, suppose that we have a sample $X_1, \dots X_n$ with $X_i \sim \mathcal{N}(b, \sigma^2)$ for $i \in \{1, \dots, n\}$ and $b \in \mathbb{R}$. If we want to test under the null hypothesis whether $H_0 : b = c$ (for $c \in \mathbb{R}$) against $H_a : b \neq c$, we define the following Z statistic as

$$Z = \frac{\bar{X}_n - c}{\sigma/\sqrt{n}}$$

and, under the very same null hypothesis, it can be shown that (you will have to trust me on this) $Z \sim \mathcal{N}(0, 1)$, that is Z is standard normal. So if the realised value of the test statistic, say z, is in absolute value greater than the $1 - \alpha/2$ quantile of the law $\mathcal{N}(0, 1)$, then we reject the null hypothesis that $b = c$. So really, the distribution of the statistic at hand governs the choice of the law that we use for the quantile.

In an applied exercise, suppose that you have as a sample available to you the x column of `dataframe` and that one of your friend tells you that this sample comes from a $\mathcal{N}(0.5, 1)$ law. You want to test whether the mean is really 0.5. So you use the test statistic

$$\frac{\bar{X}_n - 0.5}{s_X/\sqrt{n}}$$

and you compare it to a standard normal critical value with, say, $\alpha = 5\%$:

```
meanx <- mean(dataframe$x)
sdx <- sd(dataframe$x)
(tstat <- meanx-0.5/(sdx/sqrt(length(dataframe$x)))) #compute the test statistic
```

```
## [1] -4.067921
```

```
(qnorm(0.975, mean=0, sd=1))
```

```
## [1] 1.959964
```

Recall that, with $\alpha = 5\%$, we divide by two this tail area to have 2.5% in the upper tail and 2.5% in the lower tail. We see that the test statistic is, in absolute value, above to the critical value. We can confidently reject the null hypothesis saying that the mean is 0.5. In fact, I generated x as a $\mathcal{N}(0, 1)$ sample.

Question c)

Explain, in your own words, what the SST, the SSR and the SSE refer to.

The Total Sum of Squares (SST) is the "natural" variation of the response with respect to its average. It can then be decomposed into two parts: the variance explained by the model that we fit for this response, which is the SSE and the variance that is not explained by our model, the SSE. Naturally, a good (linear) model would see the SSR to be relatively high with respect to the SST and an SSE to be relatively small. This is why we use the coefficient of determination, R^2, to (partly) assess the model quality.

To access these figures in `R`, use the `anova()` command on a linear fit, such as `mod` that we fit earlier:

```
anova(mod)
```

```
## Analysis of Variance Table
##
## Response: dataframe$x
##             Df Sum Sq Mean Sq F value    Pr(>F)
## dataframe$z  1 15.924  15.924  31.719 9.103e-07 ***
## Residuals   48 24.098   0.502
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The `Sum Sq` relating to `dataframe$z` is the SSR; the `Sum Sq` relating to `Residuals` is the SSE. The sum of the two gives you the SST.

Question d)

Why does one care about constructing confidence intervals?

Confidence intervals are crucial in statistics in that they are considered more "robust" than simple point estimates. We build a confidence interval around an estimate to be able to say that, with a certain confidence level, the true parameter lies in that interval. It gives an idea about the whereabouts of a population parameter around an estimate.

For illustration, we can construct a confidence interval around the mean of the x column in `dataframe`. Suppose we know that the standard deviation of x is 1 and that x is normally distributed (at least the random process behind it). Then to construct the 95% confidence interval, one uses:

```
(CI95 <- c(mean(dataframe$x)-qnorm(0.975, 0, 1)*sd(dataframe$x)/sqrt(length(dataframe$x)),mean(datafram
```

```
## [1] -0.40639489  0.09461448
```

What it says is that 95% of the confidence intervals we build as above will include the true mean of the random process behind x.

Question e)

What is multicollinearity and how to detect it? Why is it a problem?

Multicollinearity happens when the predictors in a linear regression are correlated with each other. This "contamination" can cause the estimated coefficients to be inconsistent, for example their signs could be counter-intuitive. If one has a background in Linear Algebra, one might understand how this could contaminate the computations of the estimates, since in the X matrix storing the multivariate observations (what we call the design matrix), some columns could be, in the extreme case, linear combinations of the others and therefore cause troubles in the estimation routine of the b_is. But this is beyond the scope of this course.

To test multicollinearity, one can use the VIF as covered in the class. There is a way to compute it directly, using the `vif()` command from the `car` package in `R`. For example, suppose we run a regression with x and z to predict y, we have:

```
library(car)
```

```
## Loading required package: carData
```

```
##
## Attaching package: 'car'
```

```
## The following object is masked from 'package:dplyr':
##
##     recode
```

```
mod2 <- lm(dataframe$y~dataframe$x+dataframe$z)
vif(mod2)
```

```
## dataframe$x dataframe$z
##    1.660815    1.660815
```

Question f)

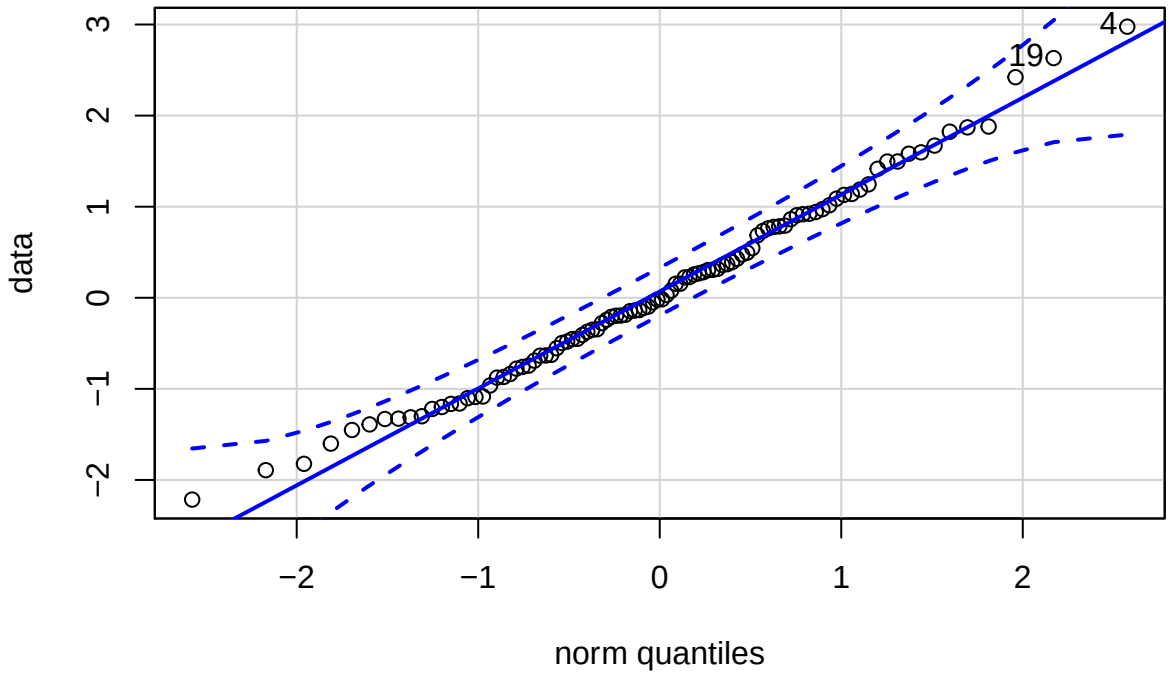
What is a QQ-plot and why is it useful?

For example, the following code generates 100 observations that are drawn from a standard normal distribution:

```
data <- rnorm(100)
```

One could then use the `qqPlot` function from the `car` package to check whether the above sample follows a standard normal distribution, by using:

```
library(car)
qqPlot(data, distribution="norm")
```



The fit in this case seems to be pretty good and there is little reason to doubt about the normality assumption. The potential departure in the tails should not be of great concern.