Introduction to R, RStudio and R Markdown

I. Zin, L. Crochemore, A. Gautheron, A. Haruna, B. Hingray, P. Séchet

This training on the R programming language, the RStudio environment and on R Markdown documents shall be completed before the first BE of Engineering Hydrology on Oct 12th.

Indicative working time: max 10 hours

Due date: Oct 11th. Drop your modified .Rmd file with your answers to instruction 40 on Chamilo in "Travaux" under "Introduction to R".

General introduction

This tutorial is a brief introduction to the **R programming language**, the **RStudio free software environment** for statistical computing and graphics, and the **R markdown notebook** environment which allows combining text and results from source codes in a single document. The objective is to become autonomous (or almost) and to be able to use R for the main basic data processing you will need to perform in the rest of the Engineering hydrology module, but also further in other modules and hopefully in your professional career. Why should you use R for your work? Hereafter, you will find the main benefits of R and RStudio:

- they are complete and widely used programming language and statistical package, both in the academic and the industrial environments. They contain efficient functions and data structures for data analysis, considered as a standard in statistics and data mining;
- they are open source software that can be downloaded for free by anyone from different mirror sites around the world and available for all OSs. This means that anyone can help developing new packages and features, leading to a fast growing number of analysis packages;
- they also contain powerful graphics.

If you have questions that are not covered by this tutorial, online manuals can be found at http://www.r-project.org under the Document and manuals menu. The 'Site du zero' also provides a really good introduction to R programming (in French, can be automatically translated using Google Translate).

Moreover, R is very well documented with help files (see next sections) and many presentations on R and RStudio are available on the web.

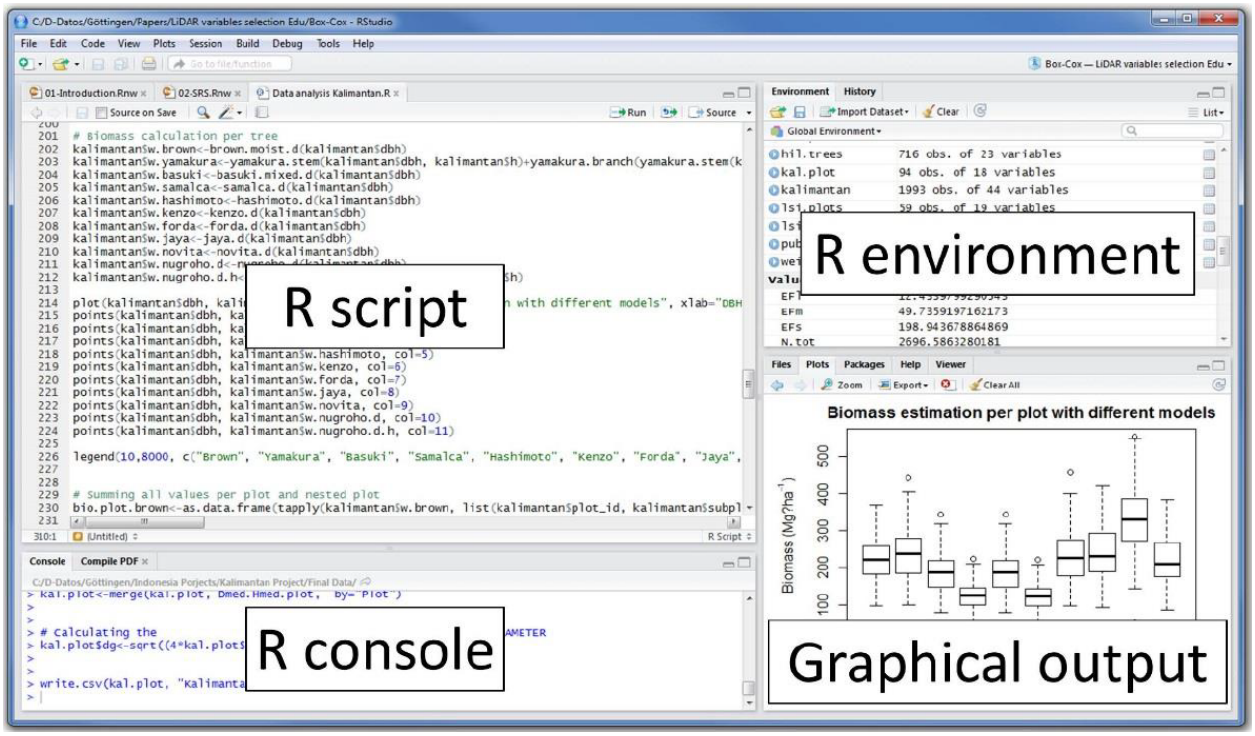
R and RStudio packages installation and use

R and RStudio are already installed at ENSE3 and available from the Program files menu. Alternatively, one can start them by double-clicking on their icons (under Windows and Mac OS X) or by typing 'R' or 'RStudio' in a terminal (default under Linux).

If you need to download and install R and RStudio on your own computer, go first to www.r-project.org for installing R. Select CRAN from the menu on the left side of the page. On the next page, select a mirror from which to download the package from and the OS (Windows, Mac, Linux) type you are using. Double click on the file you downloaded and follow the instruction to complete your installation. Then, go to https://www.rstudio.com/products/rstudio/ for downloading and installing the open source licence of RStudio (which is called RStudio Desktop).

The basic way to work with R is using a simple console (RGui), but different working environments (IDEs) exists for R. Some of them have a support for syntax highlighting and utilities to send code to the RGui: a good choice for beginners and developers is to use RStudio, as we will do in the Engineering hydrology framework.

In RStudio, you can both write your code directly within the RGui (or R console, bottom left in the following figure) or upload it within the R Editor window (top left), which is more suitable for longer scripts and only appears if you use such scripts. Two more windows allow you to get information about the datasets and variables you are using (R Environment window, top right) and to plot results, get help about R functions or get new packages (R Files/Plots/Packages/Help window, bottom right).



Exploring R: basic syntax

Instruction 1 Start RStudio.

The last line of the RGui includes the `>` prompt. This is where you enter commands that tell R what you want it to do. For instance, the return path of the current working directory is given by:

```
getwd()
```

To change the current working directory, you can use the command: `setwd("Add_your_directory_path_name_here")`. For instance: `setwd("C:/Users/toto/Desktop")` sets `C:/Users/toto/Desktop` as the new working directory.

The return content of your current working directory is obtained by: `dir()`

Instruction 2 Get your current working directory and its content, then create a working directory in `Z:` and set it the new working directory.

In R, entities such as data or functions are known as objects of different modes. You can create an object with the assignment operator `<-` (use the minor and minus keys): `object <- ...` For instance:

```
a <- 5
```

creates a numeric (integer) value : a = 5

```
x <- 0:5
```

Creates a vector of numeric (integer) values from 0 to 5 : x = [0 1 2 3 4 5]

```
z <- c(2.3, 3.4, 9.0)
```

creates a vector of numeric (floating) values : z = [2.3 3.4 9.0]

```
X <- c("a","b","c","d")
```

creates a vector of character values : X = [a b c d] The `c()` function combines its arguments. In the previous commands, it is used as a default method to form vectors. Typing the name of an object returns the object content:

```
a
```

```
## [1] 5
```

Note that R is case sensitive, so `x` and `X` are different symbols and refers to different objects.

Instruction 3 Create the above vectors in your RGui, then print the content of vectors z and X (look also at the Environment window in order to learn how to read its information content).

The mode of an object is the basic type of its fundamental constituents. It can be obtained either by looking to the Environment window or by the following command: `mode(object)` Indeed, in our example:

```
mode(a)
```

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

```
mode(mode)
```

```
## [1] "function"
```

returns the type `function`, as `mode()` is a function of R.

An object size is obtained by `length(object)`.

Instruction 4 Get the type of `x`, `z` and `X`, then evaluate their sizes.

Subsets of the elements of a vector may be selected by appending to the name of the vector an index vector in square brackets `[ ]`.

Instruction 5 Extract the second element of `y` and the fourth element of `X`.

Elementary commands consist of either expressions or assignments. If an expression is given as a command, it is evaluated, printed and the value is lost:

```
3*x
```

```
## [1]  0  3  6  9 12 15
```

does not store the data into an object.

An assignment also evaluates an expression and passes the value to an object, but the result is not automatically printed:

```
y <- 3*x
```

creates a vector y = [0 3 6 9 12 15] (please notice that operations applied to vectors are performed element by element).

Instruction 6 Create a new vector t = [3.2, 1]. Evaluate `2*x + y + 1` and `x/4+t`. What happens when you evaluate `5*t + 2*z`? Explain your results.

Instruction 7 What is the mean of the values of the `x` vector (type `mean(x)` in the prompt)? What is the mean of `z`? What happens when you apply the `mean()` function to the vector `X`? Explain your results.

In addition to numerical vectors, R allows manipulation of logical quantities. The elements of a logical vector can have the values `TRUE`, `FALSE` and `NA` (for "not available" = missing data – cf. also section 5). Logical vectors are generated by conditions. For instance: `lv <- x > 3` sets `lv` as a vector of the same length as `x`, with values `FALSE` corresponding to elements of `x` where the condition `x > 3` is not met and `TRUE` where it is.

Instruction 8 Create a logical vector generated from the `y` vector, containing `TRUE` in correspondence to the elements of `y` divisible by 2 and `FALSE` elsewhere.

For listing all the objects in the current R session type:

```
ls()
```

```
## [1] "a" "x" "X" "y" "z"
```

or

```
objects()
```

```
## [1] "a" "x" "X" "y" "z"
```

To remove objects, the function `rm()` is available: `rm(object_name)`

Instruction 9 List the objects you have created and check your result by looking in the Environment window. Then remove vector `t` and verify that `t` has been removed by listing the remaining objects.

At the end of each R session you are given the opportunity to save all the currently available objects. If you indicate that you want to do this, the objects are written to a file called `.RData` in the current working directory, and the command lines used in the session are saved to a file called .Rhistory. When R is started at later time from the same directory, it reloads the workspace from this file. At the same time the associated commands history is reloaded. Thus, it is recommended that you should use separate working directories for different analyses conducted with R.

Also, you can clear the workspace (don't do it now!) before running your scripts: `rm(list=ls())` (removes all the objects of a workspace). All objects created during an R session can also be stored permanently in a file for use in future R sessions. The function `save.image(file="myfile.RData")` allows to save the contents of workspace into a file named `myfile.RData` (or whatever name you give to your `.RData` file).

Instruction 10 Save the objects of your current R session into a file called `MyFirst_R.RData`. Save the objects `y` and `X` into a file called `yX.RData`.

Instruction 11 Quit RStudio.

For restoring previously saved values to the user's workspace use the function `load("Filename.RData")`.

Instruction 12 Restart RStudio and look at the Environment window. Do you have already loaded objects? Why?

Instruction 13 Clear the workspace and look at the Environment window.

Instruction 14 Load your `MyFirst_R.RData` file and look at the Environment window.

Specific R objects

Three important objects in R are arrays, lists and data frames.

Arrays

Matrices (2-dimensional vectors) or arrays (k-dimensional vectors, with k > 2) are multi-dimensional generalizations of vectors, which can be indexed by two or more indices. The array() function applied to a given vector (i.e. array(data_vector, dimension_vector)) allows to create matrices and k-dimensional arrays. As an example: the values contained into a vector a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] can be reordered as a 6x2 or a 2x6 matrix:

```
a <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12) # Creating a vector a of 12 elements
a
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
mat1 <- array(a, c(6,2)) # Arranging vector a of 12 elements into 6 lines and 2 columns
mat1
```

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

```
mat2 <- array(a, c(2,6)) # Arranging vector a of 12 elements into 2 lines and 6 columns
mat2
```

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

If the size of the considered vector `a` is exactly the dimension_vector size (`c(6,2)` or `c(2,6)`) (in our example 12 = 2x6 = 6x2), each value of data_vector is considered once for constructing the array. However, if `a` was shorter than the dimension_vector, its values are recycled from the beginning again to make it up to size of `a`.

Instruction 15 Create the above matrices `mat1` and `mat2`. Show their content.

Instruction 16 Imagine what the result of the following commands is, then check your answers:

- `mat3 <- array(a, c(3,2,2))`
- `mat4 <- array(a, c(2,3,2))`
- `mat5 <- array(a, c(2,2,3))`
- `mat6 <- array(a, c(3,3,2))`
- `mat7 <- array(0, c(3,3,2))`

The dimensions of an object are given by the function `dim()` applied to a given object: `dim(object)`. For instance,

```
dim(mat1)
```

```
## [1] 6 2
```

(for 6 lines and 2 columns)

Instruction 17 Check the dimensions of matrices `mat2` to `mat7` (look also at the Environment window).

Individual elements of an array may be referenced by giving the name of the array followed by the subscripts in square brackets, separated by commas. Negative indices are not allowed. NA and zero values are allowed: rows in the index matrix containing a zero are ignored, and rows containing a NA produce a NA in the result.

More generally, subsections of an array may be specified by giving a sequence of index vectors in place of subscripts; however if any index position is given an empty index vector, then the full range of that subscript

is taken. As an example:

```
mat1[1,]
```

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

gives the elements in the first line of `mat1`.

Instruction 18 What is the output of the following expressions? Explain your results.

- `mat1[1,2]`
- `mat2[1,2]`
- `mat1[3,]`
- `mat2[3,]`
- `mat1[,2]`
- `mat1[,]`
- `mat6[„]`
- `mat6[„1]`
- `mat6[,1,]`
- `mat6[1„]`

Instruction 19 Generate a 4 by 5 array `Z` containing integer values from 1 to 20. Extract elements `Z[1,3]`, `Z[2,2]` and `Z[3,1]` as a vector structure and replace these entries in the array `Z` by zeroes.

Correction

```
Z <- array(1:20, dim=c(4,5)) # generates a 4 by 5 array.
Z # prints the array values.
i <- array(c(1:3,3:1), dim=c(3,2)) # i is a 3x2 index array.
i
Z[i] # extracts the elements in the subscript array.
Z[i] <- 0 # replaces those elements by zeros.
Z
```

For those who want to manipulate vectors and matrices after this tutorial, two other exercises (Ex.1 and Ex.2) are given in the `AdditionalExercises_R.pdf` file in Chamilo.

Lists

Lists are a general form of vector in which the different elements (known as their components) don't need to be of the same type and are often vectors or lists themselves. Here is a simple example of how to make a list:

```
Lst <- list(name="Fred", wife="Mary", no.children=3, child.ages=c(4,7,9))
Lst
```

```
## $name
## [1] "Fred"
##
## $wife
## [1] "Mary"
##
## $no.children
## [1] 3
##
## $child.ages
## [1] 4 7 9
```

New lists may also be formed from existing objects. For instance, with the assignment:

```
Lst2 <-list(data1=x, data2=X, data3=y)
Lst2
```

```
## $data1
## [1] 0 1 2 3 4 5
##
## $data2
## [1] "a" "b" "c" "d"
##
## $data3
## [1]  0  3  6  9 12 15
```

Instruction 20 You will set up a list of 3 components using your vectors `x`, `X` and `y` and giving them names as specified by the argument names that can be freely chosen (here: `data1`, `data2`, `data3`). Create the above lists and look at their components. Please note that you can unroll the arrow near the list names in the Environment window of RStudio in order to have information about the list components.

The function `length()` applied to a list gives the number of components it has.

Instruction 21 How many components do the lists `Lst` and `Lst2` contain?

You can display the names of a list with the function `names()`. For us:

```
names(Lst)
```

```
## [1] "name"        "wife"        "no.children" "child.ages"
```

and

```
names(Lst2)
```

```
## [1] "data1" "data2" "data3"
```

If names are omitted, the components are numbered only. The components of a list may be individually referred to as `listname[[ ]]`. In our case: `Lst2[[1]]`, `Lst2[[2]]` , `Lst2[[3]]` refer to the first, second and last component of list `Lst2`, respectively.

```
Lst2[[1]]
```

```
## [1] 0 1 2 3 4 5
```

```
Lst2[[2]]
```

```
## [1] "a" "b" "c" "d"
```

```
Lst2[[3]]
```

```
## [1]  0  3  6  9 12 15
```

As our list components are vectors, you can access each vector entry with squared brackets, as usual:

```
Lst2[[3]][2] #gives the 2nd element of the 3rd component
```

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

If the components of a list are named, the component may be referred to either by giving the component name as a character string in place of the number in double square brackets, or, more conveniently, by giving an expression of the form `listname$component_name`. This is a very useful convention as it makes it easier to get the right component if you forget the number. In the `Lst` example given above: `Lst$name` is the same as `Lst[[1]]` and is the string `"Fred"`, `Lst$wife` is the same as `Lst[[2]]` and is the string `"Mary"`, `Lst$child.ages[1]` is the same as `Lst[[4]][1]` and is the number 4.

Instruction 22 Read the elements of each component of the list `Lst2` using both the numbers within double square brackets and the component names.

Correction

```
Lst2[[3]] # (elements of the 3rd component of list Lst2)
# gives the same result as
Lst2$data3 # (data3 is the name of the 3rd component of list Lst2)
Lst2[[1]][2] # (element of the 1st component of list Lst2 in position 2)
# gives the same result as
Lst2$data1[2]
```

Data frames

Data frames are lists having matrix-like structures, in which the columns can be of different modes and attributes. There are restrictions on lists that may be made into data frames, namely

- the components must be vectors (numeric, characters, or logicals), factors, numeric matrices, lists or other data frames ;
- vector structures appearing as variables of the data frame must all have the same length and matrix structures must all have the same row size. Indeed, a data frame may be displayed in matrix form and its rows and columns extracted using matrix indexing conventions.

Let's look to a preloaded dataset in R to see how a data frame works: the `iris` dataset.

Instruction 23 Print the `iris` dataset.

The left column displays the observation number (1 to 150) of the dataset. The remaining data in the five columns on the right are the actual data in the dataset. You can see that you have both numbers (columns 1 to 4) and text (column 5) as data. The first line gives you the name of the data: column 2 contains `Sepal.Length`, column 3 contains `Sepal.Width`, column 4 contains `Petal.Length`, column 5 contains `Petal.Width` and column 6 contains `Species`.

All objects in R have a class, reported by the function `class()`. For simple vectors this is just the mode, for example `numeric`, `logical`, `character` or `list`. `matrix`, `array` and `data.frame` are other possible values. A special attribute known as the class of the object is used to allow for an object-oriented style of programming in R. Indeed, if an object has class `data.frame`, it will be printed in a given fixed way, the `plot()` function will display it graphically in a given fixed way, and other functions will react to it as an argument in a way sensitive to its class.

Instruction 24 Find the class of the `iris` dataset. What are its dimensions?

Correction

```
class(iris)
dim(iris)  # iris is a data.frame with 5 components (columns) of 150 elements (lines)
```

As for lists, you can retrieve the name of the different components of a data frame with the `names()` function and access to the different data values using the name of the different columns.

Instruction 25 Retrieve the names of the different components of the iris dataset and access to some elements.

Correction

```
names(iris) # prints the names of the different components (here, different datasets) of iris
iris$Species # prints the 150 values of the Species dataset
iris$Species[83] # prints the 83rd element of the Species dataset
iris$Petal.Length[83:97] # prints the elements 83rd to 97th of the Petal.Length dataset
```

Any list whose components conform to the restrictions of a data frame may be coerced into a data frame using the function `as.data.frame()`.

Instruction 26 Try to coerce the lists `Lst` and `Lst2` into data frames. Print and examine the results.

Correction

```
as.data.frame(Lst) # creates a data.frame with 4 columns (the number of components of list Lst) and 3 li
as.data.frame(Lst2) # returns `Error in data.frame(data1 = 0:5, data2 = c("a", "b", "c", "d"), data3 = c
```

You cannot create a data.frame from lists having a different number of elements in each component (for Lst2 : data1 has 6 elements, data2 has 4 elements, data3 has 6 elements) – see intro of this section 4.3.

Some useful R functions

Files called "R reference card" and "r_cheasheet" are available in the BE folder in Chamilo. They contain the basic functions in R. A very short insight is given in this section. Other functions (essentially statistical) will be given in the next tutorials (TD).

Functions in R are stored within packages. If you want to know what the already installed packages are, you can use the following function: `library()`

There are various external packages that you can add to the base R package in order to add complementary specific functions. For doing that, first select the "Package" menu in the File/Plots/Packages/Help window and select "Install package(s)". Then select your desired CRAN mirror in the list that appears and select the desired package. R will download the required files and install them on your computer. In order to use the new features you have just installed, you can use the `require()` or the `library()` commands with the exact name of the desired package. For instance, if you want to use the package named "stats" that contains a number of statistical functions, you will print `library(stats)`.

The `help()` function or the `?` command gives you information about a given function: `help(function_name)` or `?function_name`.

The elementary arithmetic operators of R are the usual `+`, `-`, `*`, `/` and `^` for raising to a power. In addition, all of the common arithmetic functions are available: `log()`, `exp()`, `sin()`, `cos()`, `tan()`, `sqrt()`, etc. have their usual meaning.

Instruction 27 Assign `exp(1)` to `toto`. Evaluate `log(toto)` and `log10(toto)`. What is the difference between `log()` and `log10()`? Use `?log` to view a detailed description of the `log()` and `exp()` functions and their usage in the Help window.

The logical operators are `<`, `<=`, `>`, `>=`, `==` for exact equality and `!=` for inequality. In addition, if `expression1` and `expression2` are logical expressions, then `expression1 & expression2` is their intersection ("and"), `expression1 | expression2` is their union ("or"), and `!expression1` is the negation of expression 1.

The function `is.na()` gives a logical vector of the same size as the vector to which it is applied, with value `TRUE` if and only if the corresponding element in the vector is `NA`.

`max()` and `min()` select the largest and smallest elements of a vector, respectively.

`range()` is a function whose value is a vector of length two, containing the min and the max value of the vector to which it is applied (namely: `c(min(), max())`).

`sum()` gives the total of the elements in a vector and `prod()` their product.

`sort()` returns a vector of the same size as the one on which `sort()` is applied, with the elements arranged in increasing order. By default, `sort()` omits any `NA`.

`rank()` returns the sample ranks of the values in a vector. `order()` returns the sample ranks of the values in a vector, placing NAs last. `x[order(x)]` returns the x vector arranged in increasing order, with NAs last. `rev()` reverses the order of vector elements

The main statistical descriptors are given by:

- `mean()` that calculates a sample mean

- `median()` that calculates a sample median
- `var()` that calculates the variance
- `sd()` that calculates the standard deviation.
- `summary()` that gives the main statistical descriptors of a sample (min, 1st quartile, median, mean, 3rd quartile, max)

Instruction 28 Use the function `summary()` for calculating the main statistical descriptors of your vector `x`, your list `Lst` and the `iris` data frame. Look at the results and comment the output differences.

The function `sapply()` takes as arguments a data frame and the function that is to be applied to all columns of the data frame.

Instruction 29 Calculate the mean, the principal quantiles and the range of all the elements of the iris data frame

Correction

The 5th column of iris contains non numerical values, so we consider only the first 4 for calculating statistical parameters:

```
sapply(iris[1:4], mean)
sapply(iris[1:4], quantile)
sapply(iris[1:4], range)
```

In R, the class `Date` represents calendar dates. Different functions allow working with dates in basic R: `as.Date()` and `format.Date()` can be particularly useful (see also their help). Example :

```
# read in date/time info in format 'm/d/y'
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
as.Date(dates, "%m/%d/%y")
```

```
## [1] "1992-02-27" "1992-02-27" "1992-01-14" "1992-02-28" "1992-02-01"
```

Instruction 30 What is the output of the following assignments and expressions?

- `today <- Sys.Date()`
- `format(today, "%d %b %Y")`
- `tenweeks <- seq(today, length.out=10, by="1 week")`
- `weekdays(today)`
- `months(tenweeks)`

Instruction 31 What is the output of the following assignments and expressions?

- `as.Date(34519, origin = "1904-01-01")` (1904-01-01 is the default origin in MAC Excel)
- `as.Date(35981, origin = "1899-12-30")` (1899-12-30 is the default origin in Windows Excel)

Base graphics

R offers a number of plotting possibilities.

Instruction 32 To see some of the plotting possibilities of R, enter `demo(graphics)` and press the `Enter` key to move to each new graph.

The function `plot()` allows plotting one vector against another (the two vectors must be the same length). Several options exists for defining the parameter settings of your plot (see also the help page of the `plot()` function).

Instruction 33 Try:

- `plot((0:20)*pi/10, sin((0:20)*pi/10))` and look at the result in the plot window of RStudio.
- `plot((0:20)*pi/10, sin((0:20)*pi/10), type="l")`
- `plot((0:20)*pi/10, sin((0:20)*pi/10), type="p")`

- `plot((0:20)*pi/10, sin((0:20)*pi/10), type="b")`

Instruction 34 What is the role of the following graphical parameters `cex`, `col`, `xlab`, `ylab`, `col.lab`, `pch`?

- `plot((0:20)*pi/10, sin((0:20)*pi/10), type="b", cex=2)`
- `plot((0:20)*pi/10, sin((0:20)*pi/10), type="b", cex=2, col="red")`
- `plot((0:20)*pi/10, sin((0:20)*pi/10), type="b", cex=2, col="red", xlab="X-axis title")`
- `plot((0:20)*pi/10, sin((0:20)*pi/10), type="b", cex=2, col="red", xlab="X-axis title", ylab="Y-axis title")`
- `plot((0:20)*pi/10, sin((0:20)*pi/10), type="b", cex=2, col="red", xlab="X-axis title", ylab="Y-axis title", col.lab="blue")`
- `plot((0:20)*pi/10, sin((0:20)*pi/10), type="b", cex=2, col="red", xlab="X-axis title", ylab="Y-axis title", col.lab="blue", pch=20)`

The parameter `mfrow` can be used to configure multiple plots on one page, with plots appearing row by row. For a column by column layout, use `mfcol`.

Instruction 35 What does the following example do? What is its output?

```
par(mfrow=c(2,2), pch=16)
library(MASS) # Package MASS contains the Animals dataset that will be used for the example
attach(Animals)
plot(body,brain)
plot(sqrt(body)^0.1, (brain)^0.1)
plot(log(body), log(brain))
detach(Animals)
par(mfrow=c(1,1), pch=1) # to come back to the default graphic mode
```

Histograms can be drawn with the `hist()` function.

Instruction 36 Use and comment the following commands:

- `hist(iris$Petal.Length)`
- `hist(iris$Petal.Length, breaks=30)`
- `hist(iris$Petal.Length, breaks=10)`
- `hist(iris$Petal.Length, breaks=5)`

Other useful plotting functions are density plots (given by the `density()` function) and boxplots (given by the `boxplot()` function) – cf. the second part of the course, starting in October, for the theoretical background:

- `plot(density(iris$Petal.Length))`
- `boxplot(iris$Petal.Length)`
- `boxplot(iris$Petal.Length, horizontal=TRUE)`
- `boxplot(iris$Petal.Length, iris$Petal.Width, iris$Sepal.Length, iris$Sepal.Width)`

Of course, all of these functions have also many other optional graphical parameters – see the corresponding help pages.

Reading and writing external data

If variables are to be held mainly in data frames, as we strongly suggest they should be, an entire data frame can be read directly from an external file with the `read.table()` function. Its simplest syntax is `Mydata <- read.table("Myfilename")`.

The input file `Myfilename` is generally organized with a first line containing the name for each variable in the data frame and a series of additional lines with as their first item a row label and then the values for each variables stored in the remaining columns (such as in the iris data frame). `read.table()` takes optionally different parameters additional to the file name that holds the data. Specify `header=TRUE` if there is an initial

row of header names (the default is `header=FALSE`). In addition, user can specify the separator character: space is used as default, alternatives are `sep=","`, `sep=";"` or `sep="\t"`. Similarly, one can choose a missing value character, which by default is `NA`, with the option `na.strings`. For instance, `na.strings="-999.9"`. If `read.table()` detects that lines in the input file have different numbers of fields, data input will fail, with an error message. It is the useful to use the function `count.fields()` to report the number of fields (columns) that were identified on each separate line of the file.

Useful variants of `read.table()` are `read.delim()` and `read.csv()` – the latter having set for files that have been generated from Excel spreadsheet.

Instruction 37 Get information on the `read.table()` and `read.csv()` functions.

Instruction 38 Read the `AIR_GHG.csv` file given in Chamilo. Note that there is a header and that the separator is "," (comma). How many variables are stored within the `AIR_GHG.csv` file? What are their names? How long are the datasets?

Similarly, one can write a data frame within an external file using the `write()` or `write.table()` or `write.csv()` functions.

Instruction 39 See the help pages of such writing functions for further details.

If you want to strengthen your R knowledge after this tutorial, try to solve Ex.3 of the `AdditionalExercises_R.pdf` file in Chamilo.

R markdown introduction

This document was entirely written in R and is compiled from an R Markdown Notebook named `BE0.Rmd`. R Markdown notebooks allow combining formatted text, R code and figures and tables produced with R in a single document. R Markdown also allows for document formatting with, for instance, a table of content, figure and table captions, links to websites. R Markdown allows creating reports, but also slide presentations and websites if needed.

R Markdown documents have an extension `.Rmd`. Such scripts can combine standard text, called "in-line text", and "code chunks" which allow inserting command lines and command line outputs in a document. In-line text is written directly in the `.Rmd` file (see for example this paragraph in `BE0.Rmd`). Different types of "code chunks" allow inserting code, figures or data in the document. You will get to test these below. Within these chunks, any R code can be placed, and all commands seen in the above tutorial can be run within a notebook. When you execute code within the notebook, the results appear beneath the code once you "knit" (compile into an output document in either HTML, PDF or Word document).

Lessons 1 to 10 available here provide the necessary information needed to write a notebook (this website was created using R Markdown!). The file `rmarkdown_cheatsheet` in Chamilo summarizes the main commands to write notebooks and format texts and figures.

Instruction 40 Follow the instructions below by modifying the code in `BE0.Rmd` when needed. Questions 4-9 focus on creating chunks, questions 10-13 focus on in-line text, and questions 1-3 and 14-17 focus on the RStudio environment for R Markdown.

1. Open RStudio and install the package `rmarkdown`.
2. Save `BE0.Rmd` and `rstudio.png` files in a folder of your choice on your `Z:` drive, and open `BE0.Rmd` in RStudio.
3. Preview the compiled version of this document by clicking on the `Preview Notebook` button. This creates a file named `BE0.nb.html` in the same folder as `BE0.Rmd`. A window should open, showing the rendered (compiled and with text formatting) document.
4. Execute this chunk by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Ctrl+Shift+Enter*.

```
1+2
```

5. Copy and execute the above chunk adding the chunk option `, echo=FALSE` in the chunk brackets `{r}`. Preview and observe the difference.
6. Copy and execute the above chunk adding the chunk option `, eval=FALSE` in the chunk brackets `{r}`. Preview and observe the difference.
7. Add a new chunk by clicking the *Insert Chunk* button in the *Code* drop-down menu, by copy-pasting the above chunk or by pressing *Ctrl+Alt+I*. In this new chunk, create a vector of dates of length 9 and centered on today's date. Display the vector.
8. Add a chunk in which you will create a vector named 'student_nb' made of each of the digits in your student number. Write lines to display `TRUE` if the sum of this vector is even, and `FALSE` if the sum is odd.
9. Add a chunk and create a plot showing the sin(x) function applied to the interval [0,10] with a precision of 0.1. Label the axes 'x' and 'sin(x)'. Add an adequate title using the chunk option `fig.cap='your title'`.
10. Add your last name in normal text below.
11. Add your first name in italic text below (for this question and the following three, use the R Markdown cheatsheet, section Write with Markdown).
12. Write in bold text below what you want to learn during this Engineering Hydrology course.
13. Add a link named 'HERE' sending to https://ense3.grenoble-inp.fr/
14. Change the author names of the document (at the beginning of this file) to add your names in the author field.
15. Add a table of content by going into the Output options > General > Include table of contents.
16. Save your work. When you save the notebook, an HTML file containing the code and output will be saved alongside it (used for the *Preview*). The preview shows you a rendered HTML copy of the contents of the editor. *Preview* does not run any R code chunks. Instead, the output of the chunk when it was last run in the editor is displayed.
17. *Knit*, unlike *Preview*, executes each code chunk before creating either an HTML, PDF or Word document. On school computers, you will not be able to create PDF documents, but you will be able to knit to HTML or Word. Knit your document to Word. Upload your knitted `.Rmd` file including your answers to Instruction 40 in Chamilo (Engineering Hydrology > Travaux > Introduction to R).

In your reports, all commands run in R should appear (`echo=TRUE`) as well as results (`eval=TRUE`), unless you are (a) loading or installing a package, or (b) setting the path to a directory. All the text that does not correspond to R commands should be added outside chunks.

References

For further information on R commands and more advanced R Markdown document settings, you can have a look at the following links.

- **R**
  - http://www.r-project.org/manuals.html
  - https://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf
  - https://girke.bioinformatics.ucr.edu/GEN242/tutorials/rbasics/rbasics/
- **R Markdown**
  - https://rmarkdown.rstudio.com
  - R Markdown Cookbook