Introduction to 

Xavier Gendre

September 21, 2017

Contents

Acknowledgements

This document is essentially based on the two french papers written by Sébastien Déjean and Thibault Laurent,

- *Pour se donner un peu d'R*, Sébastien Déjean
  http://www.math.univ-toulouse.fr/~sdejean/PDF/un-peu-d-R.pdf
- *Encore besoin d'R?*, Sébastien Déjean and Thibault Laurent
  http://www.math.univ-toulouse.fr/~sdejean/PDF/R-avance.pdf

I deeply thank the authors for allowing me to use their great works.

1 Preliminaries

1.1 What is R?

According to *The R Project for Statistical Computing*,

> R is a language and environment for statistical computing and graphics. It is a GNU project (https://www.gnu.org/) which is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R.
>
> R provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible. The S language is often the vehicle of choice for research in statistical methodology, and R provides an Open Source route to participation in that activity.
>
> One of R's strengths is the ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control.
>
> R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License (https://www.r-project.org/COPYING) in source code form. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS.

Novice and experienced R users can consult the R project homepage,

https://www.r-project.org/

and *The Comprehensive R Archive Network*

https://www.cran.r-project.org/

They will find all the useful resources: how to install, updates, packages, FAQ, newsletter, documentation, etc.

1.2 Working with R

There are various ways to work with R: entering commands through the R prompt, writing executable scripts or using a graphical user interface (R Commander, RGui, RStudio, etc). In the sequel of this document, we will use the R prompt and give commands to illustrate the discussed topics. We make this choice for two reasons,

- whatever your operating system and the software you use to deal with R, you have a R prompt at your disposal,

- 'click-button' interfaces usually hide important things while entering the commands allow to highlight key concepts.

In the sequel, the R commands will be written in `verbatim` style. R commands to be experimented are highlighted in the following way,

```
R > help(plot)
R > print(
+ >  "Hello world!"
+ > )
```

Note the convention used for the prompt: **R** > stands for the R prompt, it does not have to be entered in the command line. The R prompt becomes + > when a command is incomplete, in such a case the command spreads over several lines until it is finished.

In R language, a comment starts with a character `#` and anything after it until the end of the line won't be interpreted. Using comments is a good habit, especially when you write scripts.

Any R function needs to be followed by a pair of brackets containing its arguments. If the function takes no argument, you still have to put an empty pair `()`. Indeed, without brackets, the function is not called but its code is displayed (this is often not what we want).

```
R > # It is a comment
R > print( # Comments can be put at the end of a line
+ >   "R is fun!")
R > q # Without brackets, show the code
R > q() # With brackets, quit R
```

1.3 Getting help

R provides a large range of help pages. They are very helpful and they constitute a good starting point when you need to use a new tool. To reach them, you can use the function `search` or the prefix `?`. As an example, the help about the function `plot` is accessible as follows,

```
R > help(plot)
R > ?plot # Shortcut version
```

Generally, the help pages consist of the following sections:

- `Description`
- `Usage`
- `Arguments`
- `Details`

- `Value`
- `Note`
- `Authors`
- `References`
- `See Also`
- `Examples`

The names of these sections are quite self-explanatory to understand what they contain. For novices, the more useful sections are the last two. Section `See Also` contains links to other functions related to the searched one, this is often fruitful to follow these links in order to get a good outlook. Section `Examples` allows to learn how to use the function by giving working commands that you can directly copy-paste.

If you do not know the name of the function to use or if you want to look for general words, you can search for a given word among all the help pages. To do this, you can use the function `help.search` or the prefix `??`. To get a lot of information about how to plot graphics with R, you can enter:

```
R > help.search("plot")
R > ??plot # Shortcut version
```

You obtain a list of functions prefixed by the name of the related package (see below) and `::` and followed by a brief description. Thus, you can call `help` to get more informations. To get more informations about `help` and `help.search`,

```
R > help(help) # Yes, it works!
R > help(help.search)
```

1.4 Packages

A R package is a set of functions (and sometimes also data) offered by the community of R users for extending the R language. Usually, a package is devoted to a specific task described by its name. At the time of writing this document, the package repository of the CRAN (*Comprehensive R Archive Network*) features 7111 available packages.

To use a function provided by a package, you need two preliminary steps:

- installation: the function `install.packages` permits to get the sources and to install a package. This step only has to be done one time for each package you need.
- loading: the function `library` loads the package in the current R environment and allows you to use its functions. This step needs to be done each time you start R and you need the package.

It is wise to keep your packages up to date. To this end, you should regularly run the command `update.packages`.

To list all the installed packages (but not necessarily loaded), simply run `library` without argument. The command `search` gives you the list of all the packages currently loaded. To get the content of a loaded package, we can call the function `ls` by giving it the index of the package in the list returned by `search`. Here is an example for the package `foreign`,

```
R > install.packages("foreign") # Install the package
R > library() # Check if the package is installed
R > library(foreign) # Load the package
R > search() # Check if the package is loaded, often in position 2
R > ls(pos=2) # List content of the package
R > update.packages() # Update installed packages
```

2 Basic concepts

This section mainly focus on manipulating the various types of elementary objects defined in R. We also broach the ways to import and export data.

2.1 Variables and types

2.1.1 Variables

```
R > a = 17
R > b <- 8
R > 17 -> c
R > a + b + c
R > z_42 <- "Hello"
R > z_42 <- a + b + c
R > ls()
R > objects()
R > rm(z_42)
R > remove(a)
```

- A variable name starts with a letter and can only contain alphanumeric characters and '_'.
- To assign a value to a variable, you can use the operators `=` or `<-`. There also exists the operator `->` for which the variable to be assigned has to be placed after the operator.
- The functions `ls` and `objects` return a list the objects defined in the current environment.
- The functions `rm` and `remove` can be used to remove objects.

2.1.2 Scalars

```
R > a <- 2 + 2
R > pi
R > cos(3*pi/2)
R > b <- exp(8.17)
R > typeof(a)
R > typeof(b)
R > typeof(a + b)
R > s <- "Hello"
R > typeof(s)
R > 2 == 3
R > t <- 2 < 3
R > typeof(t)
```

- The function `typeof` determines the R internal type of any object.
- Identify the different types of objects.

2.1.3 Vectors

```
R > v1 <- c(2, 3, 5, 8, 4, 6); v1
R > typeof(v1) # Type of elements
R > is.vector(v1)
R > c(1, 3.14, "Hello")
R > 1:10
R > seq(from=1, to=20, by=2)
R > seq(1, 20, by=5)
R > seq(1, 20, length=5)
R > rep(5, times=10)
R > rep(c(1, 2), 3)
R > rep(c(1, 2), each=3)
R > v1[2]; v1[2:4]; v1[c(1, 4)]
R > v1[-3]
R > v1[-1:2] # Error, why?
R > v1[-(1:2)]
R > v1[3] <- NA; v1
R > summary(v1)
R > is.na(v1)
R > help(NA)
R > any(is.na(v1))
R > all(is.na(v1))
R > v2 <- c(a=32, b=26, c=12, d=41)
R > v2["b"] <- 22; v2
R > names(v2)
R > names(v2) <- c(
+ >   "a1", "a2", "a3", "a4")
R > v2 > 30
R > v2[v2 > 30]
R > which(v2 > 30)
R > v2 + 100
R > v1 + v2 # Error, why?
R > 1:4 + v2; 1:8 + v2
R > cos(v2)
R > length(v2)
R > sort(v2)
```

- To create a vector we can combine its elements with the function `c`, use a sequence with `seq` (or operator `:`) or a repetition with `rep`.
- All the elements of a vector have the same type.
- To get an element of a vector via its index, we use the operator `[]`. Several indices can be passed at one time. A negative index omits the corresponding element.
- `NA` (*Not Available*) permits to deal with missing data.
- The functions `any` and `all` can be very useful when working with vectors.
- Instead of indices, we can use names.
- Conditional extraction of elements is a very useful tool when dealing with vectors.
- Various operations can be done on vectors up to some length restrictions.

Questions

1. Describe the various arguments of the function `seq`.
2. Describe the various arguments of the function `rep`.
3. What is the function `unique`? Give an example use case of this function.

2.1.4 Matrices

```
R > A <- matrix(1:15, ncol=5)
R > A; t(A)
R > B <- matrix(1:15, nc=5, byrow=TRUE)
R > B2 <- B; B2[1, 1] <- "Hello"; B2
R > typeof(B); typeof(B2)
R > cbind(A, B)
R > rbind(A, B)
R > A[1, 3]; A[2,]; A[, 2]
R > A[1:3, 2:4]
R > g <- seq(0, 1, length=20)
R > C <- matrix(g, nrow=4)
R > C[C[, 1] > 0.1,] # ***
R > A + B; A * B # Elementwise
R > A %*% t(B) # Matrix product
R > cos(A)
R > I <- diag(rep(1, 2))
R > diag(A)
R > D <- solve(A[1:2, 1:2])
R > all(A[1:2, 1:2] %*% D == I) # Why?
R > apply(A, 2, sum)
R > apply(A, 1, max)
```

- Matrices are constructed with the function `matrix`. Note that the arguments of `matrix` can be ambiguous (*e.g.* `nc` for `ncol`).
- Function `t` returns the transposed matrix.
- Like vectors, matrices contain only one type of data.
- To extract a submatrix or access to some elements, use the operator `[]`. First argument is for the row index, second one is for column index.
- Be careful with standard operators like `+` or `*`, they act element by element. The matrix product operator is `%*%`.
- Various operations can be done on matrices up to some length restrictions.
- You can create diagonal matrix with `diag`. This function can also extract the diagonal elements of a matrix.
- To inverse an invertible matrix, use `solve`.
- The function `apply` is crucial in R. This is quite the basics of this software! Moreover, `apply` is much faster than using loops.

Questions

1. What do the functions `cbind` and `rbind`?
2. Explain what happens in command `***`.
3. See the help page of `apply` and understand the two last examples.

2.1.5 Arrays

```
R > A <- array(1:12, c(2, 3, 2))
R > A
R > dim(A); length(A)
R > nrow(A); ncol(A)
R > apply(A, 1, mean)
R > apply(A, 2, mean)
R > apply(A, 3, mean)
```

- The object produced by the function `array` generalizes the matrix object. It can be seen as an array of matrices and can have more than three dimensions.
- You can use `apply` with arrays.
- It is wise to avoid naming a variable `T` or `F` in order to not confuse with `TRUE` and `FALSE`.

Questions

1. Explain the three calls to `apply`.
2. Create an array with four dimensions and compute the sums of its elements in all the dimensions.

2.1.6 Lists

```
R > l1 <- list("Bobby", 1:8); l1
R > l1[[1]]
R > l1[[2]] + 10
R > l2 <- list(
+ >   vect=1:5, text="DVORAK", scal=8)
R > names(l2)
R > l2$text
R > l2[c("scal", "vect")]
R > length(l2); length(l2$vect)
```

- An object of type `list` can contain different types of objects.
- A `list` is useful to return more than one value from a function.
- To access to an element of a list, use operator `[[]]` or `$` if the element has a name.

2.1.7 Data frames

```
R > height <- runif(20, 150, 180)
R > mass <- runif(20, 50, 90)
R > sex <- sample(c("M", "F"), 20,
+ >   rep=TRUE)
R > color <- c("Blue", "Green", "Brown")
R > eyes <- sample(color, 20, rep=TRUE)
R > table(sex); table(eyes)
R > table(sex, eyes)
R > H <- data.frame(
+ >   height, mass, sex, eyes)
R > H; summary(H)
R > head(H)
R > tail(H)
R > H[1,]
R > H$height
R > H$sex
R > is.data.frame(H)
R > is.matrix(H)
R > as.matrix(H) # Cast as a matrix
```

- Functions `runif` and `sample` will be described later. They produce random vectors.
- Notice the results returned by the function `table`.
- The function `data.frame` returns a structure devoted to handle data sets with individuals in lines and variables in columns.
- Like with the lists, the variables can have different types.
- Notice the similarities between `data.frame`, `list` and `matrix`.

Questions

1. Try the function `summary` on various types of object.
2. What is the consequence of cast of `H` as a mtrix?
3. Extract the mass of the individuals with a height greater than 160.
4. Extract the mass and the sex of these individuals.
5. Extract the height of males whose mass is less than 70. You can do it in one line with the logical operator `&` (see `help("&")`).

2.2 Input and output

2.2.1 Importing a data set

Use a text editor to create the following files,

- File *file1.csv*:

```
5,2.5,3.8
8,3.2,3.4
12,4.6,5
```

- File *file2.txt*:

```
5 2.5 3.8
8 3.2 3.4
12 4.6 5
```

- File *file3.txt*:

```
X1;X2;X3
5;2.5;3.8
8;3.2;3.4
12;4.6;5
```

- File *file4.txt*:

```
5;2,5;3,8
8;3,2;3,4
12;4,6;5
```

```
R > f1 <- read.table("file1.csv", sep=",")
R > f1
R > f1bis <- read.csv("file1.csv")
R > f1bis
R > f1bis <- read.csv("file1.csv",
+ >   header=FALSE)
R > f1bis
```

- The function `read.table` reads the content of a text file and returns a R object based on it.
- You can specify whether the file contains a header line, what is the column separator and what is the character for decimal point with arguments `header`, `sep` and `dec` respectively.
- The functions `read.csv` and `read.csv2` are similar to `read.table` but with different default parameters.

Questions

1. Import the files *file2.txt*, *file3.txt* and *file4.txt*.

2.2.2 Exporting a data set

```
R > A <- seq(1, 10, length=50)
R > write.table(A, "A.txt")
R > sink("A2.txt")
R > A
R > summary(A)
R > sink()
R > summary(A)
```

- The function `write.table` prints its required argument to a file.
- The function `sink` redirects the results of the next commands to a file instead of displaying them in the standard output. To close the file and stop sinking, simply call `sink` without argument.

2.3 Graphic functions

2.3.1 Discrete and qualitative data

```
R > v <- c(12,10,7,13,26,16,4,12)
R > pie(v)
R > pie(v, clockwise=T)
R > names(v) <- LETTERS[1:8]
R > barplot(v)
R > par(mfrow=c(1, 2))
R > pie(v); barplot(v)
R > par(mfrow=c(1, 1))
R > barplot(v, col=1:8)
R > dotchart(v)
R > par(bg="lightgrey")
R > dotchart(v, pch=16, col=1:8)
R > par(bg="white")
R > colors()
```

- The functions `pie`, `barplot` and `dotchart` (like all the other graphic functions) offer a large amount of arguments that allow you to modify the appearance of the graphic result.
- General graphic parameters can also be set through the function `par`. See `help(par)` for a lot of details on this topic.
- Notice the use of `par(mfrow=c(1, 2))` to split the graphic window into two areas (one line, two columns) and to plot several graphs in the same device.

Questions

1. What is the difference between `par(mfrow=c(2, 2))` and `par(mfcol=c(2, 2))`?
2. Through the help pages, experiment optional graphic arguments of the functions `pie`, `barplot` and `dotchart`.
3. Test and comment the following commands,

```
R > n <- 200
R > pie(rep(1, n), labels="", col=rainbow(n), border=NA)
```

2.3.2 Quantitative data

```
R > x <- rnorm(50)
R > boxplot(x)
R > hist(x)
R > stripchart(x)
```

- The function `rnorm` will be described later. It produces a Gaussian random vector.
- As other graphic functions, `boxplot`, `hist` and `stripchart` offer a lot of argument to customize your graphs.

Questions

1. Plot in the same window the 'stripchart', the horizontal boxplot and the histogram one below the other.

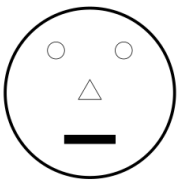
2.3.3 2D Graphics

```
R > x <- seq(-10, 10, length=50)
R > plot(x, sin(x))
R > plot(x, sin(x), type="l")
R > abline(v=0, col="blue", lwd=5, lty=3)
R > abline(h=sin(0.7), col=3)
R > text(-5, -0.5, "Hello", font=3)
R > par(mfrow=c(1, 2))
R > plot(x, sin(x), type="l", col=1,
+ >   main="Sinus")
R > plot(x, cos(x), type="b", col=3,
+ >   xlab="X-axis")
R > par(mfrow=c(1, 1))
R > plot(x, cos(x), type="l")
R > points(0, 1, pch="o", cex=3,
+ >   col="blue")
R > lines(c(-5, 5), c(0, 0), lty=2, col=2)
R > locator(1)
R > text(locator(2), c("tic","tac"),
+ >   font=c(2, 3))
R > A <- cbind(seq(0, 1, length=20),
+ >   rnorm(20), runif(20))
R > matplot(A, type="b")
```

- Some graphic functions create a new graph and others draw over an existing graph.
- The function `locator` reads the position of the graphics cursor when the first mouse button is pressed.
- The arguments `main`, `xlab`, `ylab`, ... allow to set some captions on the graph or on the axes.
- The function `matplot` plots the columns of a matrix.

Questions

1. Spend some time to understand the various graphic functions and their arguments in the above examples.
2. Draw the following guy face,



3. Give a hat to your guy.

Hint

```
R > plot(0, 0, xlim=c(-15,15), ylim=c(-15,15), type="n", axes=FALSE,
+ >   xlab="", ylab="", asp=1)
R > points(0, 0, pch=2, cex=4)
R > points(c(-4, 4), c(5, 5) ,cex=4)
R > rect(-3, -6, 3, -5, col="black")
R > lines(10*sin(0:360*pi/180), 10*cos(0:360*pi/180), lwd=5)
```

2.3.4 Headed to 3D

```
R > M <- matrix(1:100, ncol=10)
R > image(M)
R > x <- seq(-10, 10, length=30); y <- x
R > f <- function(x, y) {
+ >   r <- sqrt(x^2+y^2)
+ >   10 * sin(r)/r
+ > }
R > z <- outer(x, y, f)
R > z[is.na(z)] <- 1
R > persp(x, y, z)
R > persp(x, y, z, theta=30, phi=30,
+ >   expand=0.5, col="lightblue")
R > image(x, y, z)
R > contour(x, y, z)
R > filled.contour(x, y, z)
R > image(x, y, z)
R > contour(x, y, z, add=TRUE)
R > install.packages("rgl")
R > library(rgl)
R > x=rnorm(100)
R > y=rnorm(100)
R > z=rnorm(100)
R > plot3d(x,y,z)
```

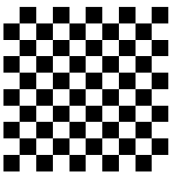
- To display 3D data, you can use `image`, `persp` and `contour`.
- Function definition will be described later.
- To understand the function `outer`, try

```
R > outer(1:5, 1:5, "+")
```

- The package `rgl` allows to get interactive 3D graphics (zoom, rotation).
- There exist other packages to enhance the standard graphic outputs (see `ggplot2` for example)

Questions

1. Draw the following 10×10 checked pattern. You can do it in just one line.



2.3.5 Exporting graphics

```
R > pdf("my_file.pdf")
R > plot(1:10, col=c("orange", "blue"))
R > dev.off() # Close the device
R > jpeg("my_file.jpg")
R > plot(1:10, col=c("orange", "blue"))
R > dev.off() # Close the device
R > png("my_file.png")
R > plot(1:10, col=c("orange", "blue"))
R > dev.off() # Close the device
```

- To export a graphic output, you need a graphics device. R provides several ones for classic format: `pdf` to export in PDF file, `jpeg` to export in JPG file, `png` to export in PNG file, etc
- When a graphics device is open, the graphic outputs no longer appear in the standard window.
- Always close the graphics device with `dev.off` when you have finish your masterpiece.
- See `library(help="grDevices")` to get a full list of graphics devices.

2.4 Programming

2.4.1 Conditionals

```
R > x <- rnorm(10)
R > if (is.double(x)) print("OK")
R > if (is.integer(x)) print("KO")
R > if (x[1] > 0) 1 else -1
R > if (x[1] > 0) {
+ >   y <- 1
+ >   print("Positive")
+ > } else {
+ >   y <- -1
+ >   print("Non positive")
+ > }
R > y <- ifelse(x > 0, 1, -1); y
R > z <- "cat"
R > switch(z,
+ >   cat=print("Hi Felix!"),
+ >   dog=print("Hi Snowy!"),
+ >   print("What is this pet?"))
```

- A conditional starts with `if` followed by a `logical` and a command to run only if the `logical` is `TRUE`.
- The `if` statement can be followed by `else` and a command to run if the initial `logical` is `FALSE`.
- When there are several commands to run, you need to gather them together in a block between `{` and `}`.
- You can abbreviate a conditional with the function `ifelse`.
- A `switch` statement chooses one of the further arguments and run the associated command.

2.4.2 Loops

```
R > x <- c(17, 8, 42, 3)
R > for (e in x) print(e)
R > for (i in 1:length(x)) print(x[i])
R > for (i in seq_along(x)) print(x[i])
R > for (i in seq_len(5)) {
+ >   fact <- prod(1:i)
+ >   cat(i, "! = ", fact, "\n", sep="")
+ > }
R > for (k in seq_len(10)) {
+ >   if (k %% 2 == 0) next
+ >   print(k)
+ > }
R > i <- 1; s <- 0
R > while (i <= length(x)) {
+ >   s <- s + x[i]
+ >   i <- i + 1
R > }
R > s
R > x <- 0
R > repeat {
+ >   print(x)
+ >   x <- x + 1
+ >   if (x == 10) break
R > }
```

- Loop statements starts with `for`, `while` or `repeat`.
- Using loops is quite always slower than using vectorial operations.
- With `for`, we iterate along a vector or an iterator. When this is possible, it is smarter to use `seq_along` and `seq_len`.
- The function `cat` concatenates its argument and display them. It is useful to get well formatted string.
- The instruction `next` halts the processing of the current iteration and advances the looping index.
- The loop `while` repeats its content until the given logical becomes `FALSE`. If the logical is `FALSE` at the beginning, the content of the loop is not evaluated.
- The loop `repeat` repeats its content until it reaches an instruction `break`. Its content is always at least evaluated one time.

Questions

1. Why is it better to use `seq_along` and `seq_len`?
2. The example of the `while` loop can be done in one line. See `help(sum)` to compute it.
3. Compute the mean of `x` with `for`, `while` and `repeat` loops. Compare your result with the value returned by `mean(x)`.
4. Display the following "ASCII art" with a loop,

```
    *
   ***
  *****
 *******
```

2.4.3 Functions

```
R > f1 <- function() print("Hello!")
R > f1
R > f1()
R > f2 <- function(k) cat(2*k)
R > f2(21)
R > f3 <- function(k) return(2*k)
R > f3(21)
R > y <- f3(21); y
R > f3 <- fix(f3)
R > f4 <- function(a, b=0) return(a + 2*b)
R > f4(2, 3); f4(5); f4(b=2, a=1)
R > f5 <- function(a, b=a) return(a + 2*b)
R > f5(2, 3); f5(5)
R > my_circle <- function(r) {
+ >   p <- 2*pi*r
+ >   a <- pi*r*r
+ >   return(list(radius=r,
+ >              perimeter=p,
+ >              area=a))
+ > }
R > y <- my_circle(3)
R > y$area == pi*y$radius^2
R > my_var <- 17
R > f6 <- function(x) {
+ >   my_var <- x
+ >   print(my_var)
+ > }
+ > print(my_var); f6(8); print(my_var)
```

- Avoid using `F` as name for a function not to be confused with `FALSE`.
- Use `function` to create a function and `fix` to edit its body.
- To return a value from a function, use `return`. Note that it ends the function and any command after `return` is not interpreted. If the end of a function is reached without calling `return`, the value of the last evaluated expression is returned.
- A default value can be given to an argument.
- A function can return only one object. If you need more, use a list.
- Variables defined in a function are limited to the scope of the function.

Questions

1. What is the difference between `f2` and `f3`?
2. Write a function `my_rectangle` that takes two arguments `l1` and `l2` and return the lengths of the sides, the perimeter and the area of the rectangle. How to deal with a square?
3. Write a function that computes the `n` first terms of the Fibonacci sequence ($u_1 = u_2 = 1$ and $\forall n > 2,\ u_n = u_{n-1} + u_{n-2}$).
4. Write a function that removes the lines of a matrix or a data frame that contain at least one `NA`.

2.5 A bit of statistics

2.5.1 Distribution

```
R > sample(c("Blue", "Red"), 10,
+ >   replace=TRUE, prob=c(4, 1))
R > help.search("Distribution")
R > help(rnorm)
R > rnorm(10)
R > dnorm(0)
R > pnorm(1.96); qnorm(0.975)
R > plot(dnorm, -3, 3, col="blue",lwd=3)
R > y <- seq(qnorm(0.975), 3, length=100)
R > polygon(c(y, rev(y)),
+ >   c(dnorm(y), rep(0, 100)), col=3)
R > text(2.2, 0.015, "0.025", cex=0.9,
+ >   font=2)
R > set.seed(17881)
R > rnorm(1)
R > set.seed(17881)
R > rnorm(1) # Wow!
```

- The most common distributions are available with R.
- For categorical distribution, the function `sample` takes a sample of the specified size from the elements of a given vector of values.
- For each defined distributions, you have at your disposal several functions. For example, the Gaussian distribution comes with `dnorm` for the density function, `pnorm` for the distribution function, `qnorm` for the quantile function and `rnorm` to generate random realizations.
- Note the way to use `plot` in the example.
- The function `set.seed` allows to set the initial seed of the random number generator. When a seed is given, it sets the "randomness" and you can reproduce the random events.

Questions

1. What are the arguments of `sample`.
2. Generate a sample of 256 independent exponential variables. In the same graphic, draw the associated histogram and the density function of the exponential distribution.
3. Plot the density function and the distribution function of several distributions (Cauchy, χ^2, ...)

2.5.2 Statistical tests

```
R > help.search("Test")
R > x <- rnorm(100)
R > y <- rnorm(100, mean=1)
R > t.test(x, y)
R > my_result <- t.test(x, y)
R > my_result
R > names(my_result)
R > my_result$method; my_result$p.value
R > var.test(x, y)
R > t.test(x, y, var.equal=TRUE)
R > cor.test(x, y)
R > ks.test(x, y)
R > ks.test(x, "pnorm")
R > ks.test(y, "pnorm")
R > ks.test(y, "pnorm", 1)
```

- The most common statistical tests (and some that are less so) are available with R.
- Use `t.test` for Student's t-tests, `var.test` for F-tests, `cor.test` for correlation tests, `ks.test` for Kolmogorov–Smirnov tests, ...
- The result of a test can be saved in a variable to use its elements later (p-value, test statistic, ...).

Questions

1. Apply the Shapiro-Wilk test instead of the Kolmogorov-Smirnov test.
2. Test the nullity of the Spearman's rank correlation coefficient between `x` and `y`.

2.5.3 Univariate and bivariate analysis

```
R > x <- runif(100)
R > mean(x); var(x); sd(x)
R > min(x); max(x)
R > quantile(x); median(x)
R > quantile(x, 0.9)
R > summary(x)
R > boxplot(x); boxplot(x, plot=FALSE)
R > my_bp <- boxplot(c(x, 2)); my_bp
R > hist(x); hist(x, plot=FALSE)
R > hist(x, density=10)
R > hist(x, nclass=5)
R > y=runif(100)
R > cov(x, y)
R > cor(x, y)
R > cor(x, y, method="spearman")
R > z <- x + rnorm(100, sd=0.05)
R > pairs(cbind(x, y, z))
```

- Functions are already defined for most common quantities to compute.
- The results of `boxplot` and `hist` can be stored without being displayed with `plot=FALSE`. It can be useful for getting some parameters of the graphics.
- You can tweak the graphic output of `hist` to suit your needs.
- The function `pairs` produces a matrix of scatterplots.

2.5.4 Linear regression

```
R > ifelse(
+ >   "package:datasets" %in% search(),
+ >   "OK", "KO")
R > help(cars)
R > res1 <- lm(dist ~ speed, data=cars)
R > res1
R > names(res1)
R > summary(res1)
R > anova(res1)
R > plot(cars)
R > abline(res1, col="red")
R > res2 <- lowess(cars$speed,
+ >   cars$dist, f=0.5)
R > lines(res2, col="blue", lty=2)
```

- First, we check that the package `datasets` is loaded. This package provides various datasets whose `cars`.
- Note the use of the operator `%in%`.
- The function `lm` fits a linear model between `cars$dist` and `cars$speed`. Note the use of argument `data`. The function `lm` can do much more than basic linear regressions.
- The first argument of `lm` is a `formula`. Such an object is used by a lot of functions in R and you can get informations with `help(formula)`.
- The object returned by `lm` can be plugged in several R functions.
- The function `lowess` uses locally-weighted polynomial regression.

Questions

1. Modify the argument `f` of `lowess` and explain what happens.

3 Advanced concepts

3.1 Specific data manipulation

3.1.1 Character strings

```
R > s <- "I play with characters"
R > length(s); nchar(s)
R > substr(s, start=1, stop=8)
R > strsplit(s, "a")
R > strsplit(s, "with")
R > s_words <- strsplit(s, " ")
R > s_words <- unlist(s_words)
R > s_letters <- strsplit(s, NULL)
R > length(s_letters[[1]])
R > toupper(s)
R > tolower("WOW!")
R > "wit" %in% s_words
R > "with" %in% s_words
R > res <- grep("play", s_words)
R > s_words[res]
R > grep("^p|c", s_words) # ***
R > grep("whit", s_words) # Search whit
R > agrep("whit", s_words)
R > paste("A", 1:5, sep="")
R > sub("play with", "master the", s)
```

- To get the number of characters in a string, use `nchar`, not `length`.
- When working with characters, you must be careful with handling of spaces.
- Note the use of the function `unlist`.
- You can search for patterns with the `grep`'s functions.
- You can use regular expressions as argument (see `help(regexp)`). For example, the command `***` looks for the words starting with 'p' or 'c'.
- The functions `paste` and `sub` are often needed by any 'Lord of the strings'.

Questions

1. What is the difference between `grep` and `agrep`?
2. In the data set `USArrests`, extract the line whose the name contains `C`. Same question for names which starts with `C`.

3.1.2 Factors

```
R > eye <- sample(c("Blue","Brown"),
+ >   size=256, replace=TRUE)
R > eye
R > eye.fact <- factor(eye)
R > eye.fact
R > eye[1] <- "Green"; eye
R > eye.fact[1] <- "Green"
R > eye.fact
R > levels(eye.fact)
R > levels(eye)
R > object.size(eye)
R > object.size(eye.fact)
R > x <- rnorm(100)
R > bins <- cut(x, breaks=-4:4)
R > bins
R > table(bins)
```

- Even if they look like strings, factors are not handled in the same way.
- A factor can only take values in a given set of levels. Giving a value that is not a valid level leads to an error and set the factor to `<NA>`.
- Factors are less bulky in memory than vectors of strings because the levels are stored only one time, each element being a reference to a level.
- The function `cut` turns a quantitative data set into a categorical data set by returning a factor object.

Questions

1. Create a vector of strings `size` with 25 elements `Small`, `Medium` or `Large`. Convert this vector into an **ordered factor** (see `help(factor)`). Verify that the levels are ordered.

3.1.3 Sets

```
R > A <- 1:10
R > B <- c(3:6, 12, 15, 18)
R > union(A, B)
R > intersect(A, B)
R > setdiff(A, B)
R > setdiff(B, A)
R > is.element(2, A)
R > is.element(2, B)
R > is.element(A, B)
R > is.element(B, A)
R > L <- letters[1:10]
R > union(A, L)
```

- Note the importance of the order of arguments in `setdiff`.
- This commands need objects with the same type (see the last example).

Questions

1. Use the operator `%in%` to achieve the same results as with `is.element`.
2. Test the membership of the letter 'k' to the vector `letters`. How can we get the position of 'k' in the alphabet?

3.1.4 Dates and time

```
R > help("Date")
R > now <- Sys.time(); now
R > today <- Sys.Date(); today
R > date()
R > weekdays(today)
R > months(today)
R > quarters(today)
R > as.Date(29813, origin="1900-01-01")
R > format(now, "%a %d %b %Y %X %Z")
R > strptime(c("03/01/1892", "02/09/1973"),
+ >   "%d/%m/%y")
R > system.time(
+ >   for(i in 1:100) var(runif(100000))
+ > )
```

- There exist specific functions to extract information about a date or a time.
- To convert an object to a date, use `as.Date`.
- To convert between character representations and date objects, use `format` and `strptime`.
- The function `system.time` return times that the command passed as argument used:

user time spent to run the command,

system time spent by the system for the command (I/O, write on disk, ...),

total sum of above times.

Questions

1. What will be the weekday of the next January 1?
2. How many days are there between now and the end of the year?

3.1.5 Merging and aggregating data frames

With the function `merge`, we can merge two data frames. It is mandatory to specify the reference column in the data frames with `by`. If the names of the reference columns differ between the data frames, we also can use `by.x` and `by.y`. For example, consider a data frame `patient` which contains information about patients and a data frame `visit` for their visits,

```
R > patient <- data.frame(
+ >   name=c("Bobby", "Cindy", "Billy", "Jenny"),
+ >   date.birth=c("1955/02/02", "1952/03/03", "1992/10/01",
+ >     "1940/02/02"),
+ >   sex=c("M", "F", "M", "F"))
R > visit <- data.frame(
+ >   patient.name=c("Bobby", "Cindy", "Bobby", "Teddy", "Billy"),
+ >   date.visit=c("2014/01/01", "2013/12/01", "2014/01/05",
+ >     "2013/12/04", "2012/10/05"))
```

The reference column is the name of the patient which has a different name in the two data frames. First, we only merge the individuals present in both data frames,

```
R > merge(patient, visit, by.x="name", by.y="patient.name")
```

If we want to keep all the informations of `patient`, we have to specify `all.x=TRUE`,

```
R > merge(patient, visit, by.x="name", by.y="patient.name", all.x=TRUE)
```

To keep all the informations of both data frames, we also have to specify `all.y=TRUE`,

```
R > patient.visit <- merge(patient, visit, by.x="name",
+ >   by.y="patient.name", all.x=TRUE, all.y=TRUE)
```

We are now interested in the age of the patients with respect to their sex at the time of their visits. We add these informations to the new data frame,

```
R > patient.visit$age <- floor(as.numeric(
+ >   as.Date(patient.visit$date.visit, format="%Y/%m/%d")
+ >   - as.Date(patient.visit$date.birth, format="%Y/%m/%d"))/365)
R > patient.visit$age
```

To get the average age with respect to the sex of patients, we need to split the data frame into subsets and compute the mean for each one. This operation in known as an *aggregation* and can be done through the function `aggregate`. We have to give a list of grouping elements with `by` and a function to apply with `FUN`,

```
R > aggregate(patient.visit$age, by=list(sex=patient.visit$sex),
+ >   FUN=mean, na.rm=TRUE)
```

Questions

1. Using the dataset `iris`, create an object `iris1` which contains the mean of `Petal.Length` for each species.
2. Create also an object `iris2` which contains the sum of `Petal.Width` for each species.
3. Merge `iris1` and `iris2`.

3.1.6 Files and directories

```
R > dir()
R > file.info(dir())
R > R.home()
R > f <- dir(file.path(R.home(), "bin"),
+ >   full.names=TRUE); f
R > f[file.access(f, 0) == 0]
R > f[file.access(f, 1) == 0]
R > f[file.access(f, 2) == 0]
R > f[file.access(f, 4) == 0]
R > dir.create("Output")
R > getwd()
R > setwd("Output/")
R > getwd() # We are in "Output" now
```

- Note the difference between `dir` and `ls`.
- Try the various arguments available for the function `dir`.
- The function `file.path` is useful for dealing with path names regardless of the operating system.
- Understand the commands with `file.access`.
- The absolute path of the current working directory is returned by `getwd`. This is the default path where R looks for the files, saves them, ... To change it, use `setwd`.
- These functions are mainly useful for writing scripts that produces tidy outputs.

Questions

1. Write a function that takes one integer argument n and does the following things:
   - generate n independent realizations of a standard Gaussian variable,
   - put the values in a file named with the current date and time in a directory called `Val`,
   - produce the boxplot of the data set in a JPEG file named similarly in a directory called `Fig`.

3.2 More programming

3.2.1 Avoiding loops

```
R > t <- array(1:24, dim=2:4)
R > apply(t, 1, sum)
R > apply(t, 1:2, sum)
R > res <- apply(t, 3,
+ >   function(x) runif(max(x)))
R > res
R > x <- rnorm(100)
R > bins <- cut(x, breaks=-4:4)
R > tapply(x, bins, mean)
R > lapply(res, mean)
R > sapply(res, mean)
R > lapply(res, quantile)
R > sapply(res, quantile)
R > v <- replicate(500, mean(rnorm(10)))
R > boxplot(v)
R > rep(x=1:4, times=4:1)
R > mapply(rep, x=1:4, times=4:1)
```

- Note that the object returned by `apply` can take many shapes.
- The sidekick of `tapply` for data frames is `by` (see `help(by)`).
- Understand the difference between `lapply` and `sapply`.
- The function `replicate` is usefull to repeat the same command a bunch of times.
- Note the object returned by `mapply`.
- These functions are core functionalities of R. They improve the readability of the code and they are faster than loops. See the following exercise to be convinced.

Questions

1. Define the two following functions,

```
R > f1 <- function(n, p) {
+ >   v <- matrix(0, n, 1)
+ >   for(i in 1:n) {
+ >     v[i] <- mean(rnorm(p))
+ >   }
+ >   return(v)
+ > }
```

```
R > f2 <- function(n, p) {
+ >   v <- replicate(n,
+ >     mean(rnorm(p)))
+ >   return(v)
+ > }
```

2. Compare the time spent in each function,

```
R > system.time(f1(50000, 500))
R > system.time(f2(50000, 500))
```

3.2.2 Advanced functions

When you define a function, you can allow to give it any argument for an other function that you want to call without having to specify all these arguments. The reserved word for that is `...`, see `help("...")` and `help(dotsMethods)`. Let's take an example,

```
R > plot.lm <- function(x, y, fit.pol=TRUE, ...) {
+ >   plot(y ~ x, ...)
+ >   abline(lm(y ~ x), col="blue")
+ >   if (fit.pol) {
+ >     pol.reg <- loess(y ~ x)
+ >     t <- seq(min(x), max(x), length.out=25)
+ >     lines(t, predict(pol.reg, t), col="red")
+ >   }
+ > }
```

Any argument of `plot.lm` apart from `x`, `y` and `fit.pol` will belong to `...` and will be passed to the function `plot` called in the second line. Try the following commands,

```
+ > x <- rnorm(100); y <- runif(100)
+ > plot.lm(x, y)
+ > plot.lm(x, y, fit.pol=FALSE)
+ > plot.lm(x, y, pch=16, col="pink",
+ >   xlab="Explanatory variable", ylab="Response variable")
```

Questions

1. Explain what the function `plot.lm` does.
2. Write a function `gaussian.hist` which takes an integer argument `n` and the reserved word `...` for arguments to be passed to `hist`. This function has to do the next steps,
   - generate `n` independent realizations of a standard Gaussian variable,
   - plot the normalized histogram of these realizations,
   - add the standard Gaussian density curve to the graph.

3.2.3 Non-interactive mode (*BATCH*)

It is feasible to run R non-interactively by giving the commands to complete through a text file. Such a way can be useful when you need to run R commands on a remote computing server. For more details, see `help(BATCH)`.

Here is an example of text file containing R commands. Let's call it *my-commands*,

```
2+2
jpeg("my_graph.jpg")
plot(rnorm(10))
dev.off()
3+3
```

To run it non-interactively from a system prompt,

- in a GNU/Linux environment,

```
$ R CMD BATCH my-commands
```

- in a Windows environment,

```
C:\> "C:\Program Files\R\R-3.1.1\bin\R.exe" CMD BATCH my-commands
```

You could also specify the name of the output file but, by default, this is *my-commands.Rout*.

Questions

1. Read `help(Rscript)` and understand the difference between `BATCH` and `Rscript`.

3.2.4 Debugging

R offers some elementary tools to debug your functions. To illustrate them, define the following three functions,

```
R > f <- function(x) return(x - g(x))
R > g <- function(y) return(y * h(y))
R > h <- function(z) {
+ >   t <- log(z)
+ >   if (t < 10) return(t^2) else return (t^3)
+ > }
```

Try to execute `f(-1)` to get an error message. Here, the problem is easy to identify but it is often more complicated with nested calls of functions and the values of arguments can be harder to follow.

To know in which function the error appears, you can call `traceback()` directly after observing the problem. It gives you the name of the last called function from top to bottom,

```
R > f(-1)
R > traceback() # Error in function h
```

Getting only the function name is often not sufficient to fix a bug. In order to know which line causes the error, you may want to step through the function using `debug`,

```
R > debug(h) # h flagged for debug
R > f(-1)
R > undebug(h) # h unflagged
```

During the call `f(-1)`, when R enters in the flagged function `h`, it breaks the run and gives you a specific prompt `Browse[2]>`. The next line to be interpreted is marked by `#N` where `N` is the step number. You can interact with R to get the values of the variables, check things, ... When you are done, validate an empty line and R goes forward to the next line to run. In such a way, you can run the function step by step and identify exactly where is the bug to fix. Use `undebug` to remove the flag and let the function runs as usual.

3.2.5 Writing scripts

Entering commands in the R prompt as we do till the beginning of this document is good for testing or for doing one-shot analysis. We often need to repeat the same operations or reuse some useful code. For that, it is possible (and encouraged) to write your R code as scripts. A R script is nothing else than a simple text file that contains R commands. You can give the file extension *.R* to your script; this is not mandatory but, for instance, if you use a text editor that does syntax highlighting, this can be useful.

Let's consider a first example, create a file *script01.R* with the following content,

```
gaussian_hist <- function(n, ...) {
  x <- rnorm(n)
  hist(x, freq=FALSE, ...)
  t <- seq(min(x), max(x), length=256)
  points(t, dnorm(t), type="l", lwd=2, col="blue")
  lines(density(x), lwd=2, lty=2, col="green")
}

# Get a color for the histogram
repeat {
  cat("Enter a color name: ")
  hist_color <- scan(file="stdin",
    what="character", n=1, quiet=TRUE)
  if (hist_color %in% colors()) {
    break
  } else {
    cat("Color \"", hist_color,
        "\" is not valid color.\n", sep="")
  }
}

# Plot the histogram in a JPG file
jpeg("my_plot.jpg")
gaussian_hist(512, col=hist_color)
graphics.off()
```

- The function `density` returns the empirical density associated to its argument.
- Note the use of `scan` to ask for user input. See `help(scan)` for more details. You could also use `readline` to this end but it only works in interactive mode.
- This script uses several techniques discussed earlier in this document. Be sure to understand all it does.

To run this script, you can use `source` in a R prompt,

```
R > source("script01.R")
Enter a color name: crimson
Color "crimson" is not valid color.
Enter a color name: goldenrod
R >
```

In a GNU/Linux environment, you can also make it executable. For that, you need to add

the shebang line `#!/usr/bin/Rscript --vanilla` in the first line of the script file and `chmod` it,

```
$ chmod +x script01.R
$ ./script01.R
Enter a color name: red
$ ls my_plot.jpg
my_plot.jpg
```

With executable scripts, you have to be careful with the R commands you use according to the non-interactive mode (see the comment above about `readline` and `scan`).

It is possible to give arguments to an executable script and to get them into R with `commandArgs`. Let's create a second example *script02.R* of executable script,

```
#!/usr/bin/Rscript --vanilla
my_args <- commandArgs(TRUE)
cat("You gave me", length(my_args), "argument(s).\n")
for (i in seq_along(my_args)) {
  cat("Argument ", i, "\t: ", my_args[i], "\n", sep="")
}
```

Then, make it executable and test it,

```
$ chmod +x script02.R
$ ./script02.R
You gave me 0 argument(s).
$ ./script02.R Hello World 42
You gave me 3 argument(s).
Argument 1        : Hello
Argument 2        : World
Argument 3        : 42
```

Note that passing arguments in such a way is not feasible through the R prompt. Thus, `source` cannot be directly used but there exist some workarounds like redefining the function `commandArgs` before calling `source` (beyond the scope of this document).

3.2.6 Calling C from R

We introduce here an elementary way to run C code into R. This topic raises many issues to address and should be treated in a more technical way. The interested reader will find more details on the internet. One of the advantages of using C functions in R is the considerable gain in execution time.

Let's create a file *convolve.c* for our C code,

```
void convolve(double * a, int * na, double * b, int * nb, double * ab) {
  int i, j;
  int nab = *na + *nb - 1;

  for(i = 0; i < nab; ++i) ab[i] = 0.0;
  for(i = 0; i < *na; i++) {
    for(j = 0; j < *nb; j++) {
      ab[i + j] += a[i] * b[j];
    }
  }
}
```

We have to compile this source code to get a proper dynamic library (**.so* in a GNU/Linux environment or **.dll* in a Windows environment) to load in our R environment,

```
$ R CMD SHLIB convolve.c
```

Thus, the easiest way to call this library from R is to load it and to create a wrapper function,

```
R > dyn.load("convolve.so") # GNU/Linux environment
R > dyn.load("convolve.dll") # Windows environment
R > convolve.with.C <- function(a, b) {
+ >   .C("convolve",
+ >     as.double(a), as.integer(length(a)),
+ >     as.double(b), as.integer(length(b)),
+ >     ab = double(length(a) + length(b) - 1))$ab
+ > }
R > convolve.with.C(1:10, seq(0, 1, length=10))
```

To get more details, you can read `help(.C)`. Note that the trailing `$ab` ensures that this function returns the created vector `ab`.

To compare the performances with a R function, let's create an equivalent function in R and measure how fast are the two versions,

```
R > convolve.with.R <- function(a, b) {
+ >   ab <- rep(0, length(a) + length(b) - 1)
+ >   for (i in seq_along(a)) {
+ >     for (j in seq_along(b)) {
+ >       ab[i+j-1] <- ab[i+j-1] + a[i]*b[j]
+ >     }
+ >   }
+ >   return(ab)
+ > }
R > system.time(convolve.with.C(1:500, seq(0, 500, length=500)))
R > system.time(convolve.with.R(1:500, seq(0, 500, length=500)))
```

The performances are clearly on the side of the function written in C. More generally, using C (or other compiled language) is a good alternative when loops are unavoidable in critical functions.

3.2.7 Parallel computing

With the multiplication of the number of CPU and their improved performances, parallel computing has known an important development in the last decade. Such an approach is no longer reserved to supercomputers and can be used to fully exploit the possibilities of your computer.

Let's take again the example of `convolve` with a new version,

```
R > install.packages("snowfall")
R > library(snowfall)
R > convolve.parallel <- function(x, a, b) {
+ >   a <- sample(a)
+ >   b <- sample(b)
+ >   ab <- rep(0, length(a)+length(b)-1)
+ >   for (i in seq_along(a)) {
+ >     for (j in seq_along(b)) {
+ >       ab[i+j-1] <- ab[i+j-1] + a[i]*b[j]
+ >     }
+ >   }
+ > return(ab)
+ > }
```

To run this function in parallel, we need to initialize a cluster with a given number of involved CPU. In the next example, we assume that you have at least 4 CPU at your disposal (if not, adjust the commands),

```
R > sfInit(parallel=TRUE, cpus=4) # Init cluster
R > system.time(result <- sfClusterApplyLB(1:4, convolve.parallel,
+ >   a=1:500, b=seq(0, 500, length=500)))
R > sfStop() # Stop cluster
```

Again, to get more details, read the help pages related to `sfClusterApplyLB`.

3.2.8 Classes

R is an object oriented language and any R object you handle is an instance of some *class*. Hereafter, we call *method* a function associated with a particular type of object. In R, you have at your disposal three object oriented systems, namely *S3*, *S4* and *R5*. Most objects in R are implemented with S3 style and we are focusing on this particular system in the sequel. If you want more informations about other systems, you will find a lot of things on the internet.

You use object oriented techniques almost all the time with R, mainly when you deal with `print`, `summary` and `plot`. These methods offer you a generic function which behaves

differently according to the class of the object. Indeed, printing a vector is differentto printing a linear regression, for instance.

```
R > my.vector <- 1:10
R > class(my.vector)
R > print(my.vector)
R > plot(my.vector)
R > my.lm <- lm(rnorm(100) ~ runif(100))
R > class(my.lm)
R > print(my.lm)
R > plot(my.lm) # Note the specific behavior of plot
```

The class of an object is given by its `class` attribute. To create an object of your custom class `MyClass`, you can use `structure`,

```
R > my.object <- structure(42, class="MyClass")
R > class(my.object)
R > my.object # Data is integer 42, class is MyClass
R > print(my.object) # Behave poorly
```

Generic functions like `print`, `mean`, `plot`, ... are usually very simple and search for a given method associated to the class of the object. For that, they use the function `UseMethod` (see `help(UseMethod)`) and fallback to a generic (and poor) way if they do not find an appropriate method (see example above for `print`). To list all available methods for an S3 generic function, or all methods for a class, use `methods`,

```
R > methods(mean)
R > methods(plot)
R > methods(t)
```

Let's see how to define a custom `print` method for objects of our class `MyClass`. Unlike other object oriented languages (C++, Java, ...), the methods are not defined in the class but according to a special naming convention `function.class` (see examples returned by `methods`). Thus, we simply need to create a function `print.MyClass`,

```
R > print.MyClass <- function(x) cat("My data is", x, ":-)\n")
R > print(my.object) # Now, it is better!
R > my.object # Silently call print
```

If you understand these mechanisms, you are now able to define your own generic function in R. As a bonus, here is an example,

```
R > my.generic <- function(x, ...) UseMethod("my.generic", x)
R > my.generic.numeric <- function(x, ...)
+ >   cat("Numeric value:", x, "\n")
R > my.generic.character <- function(x, ...)
+ >   cat("Character value:", x, "\n")
R > my.generic.MyClass <- function(x, ...)
+ >   cat("Awesome value:", x, "\n")
R > methods(my.generic)
R > my.generic(17)
R > my.generic(pi)
R > my.generic("Hello")
R > my.generic(my.object)
R > my.generic(data.frame()) # Fail, why?
```

Object oriented programming offers a tremendous amount of freedom and covering these possibilities is absolutely beyond the scope of this document. If you look for more information on the subject, you can read documents about inheritance, internal generic functions, operators, ...

Questions

1. Create a custom class for which method `plot` makes sense and write your own version of `plot` for objects of this class.

3.3 More graphics

3.3.1 Plot arrangements

In Section 2.3.3, we have seen how to use `mfrow` and `mfcol` to arrange plots on a device. Dealing with these parameters can become tricky if we want complex arrangements and `layout` provides a way to tackle that in an easier way. Be careful because `layout` is totally incompatible with `mfrow` and `mfcol`.

```
R > m <- matrix(c(2, 0, 1, 3), 2, 2,
+ >   byrow=TRUE); m
R > my.layout <- layout(m, widths=c(3, 1),
+ >   heights=c(1, 3), TRUE)
R > layout.show(my.layout)
R > x <- runif(10)
R > y <- runif(10)
R > plot(x, y, pch=16, cex=2,
+ >   col=rainbow(10))
R > plot(x, rep(1, 10), pch=16, cex=2,
+ >   col=rainbow(10))
R > plot(rep(1, 10), y, pch=16, cex=2,
+ >   col=rainbow(10))
```

- Compare the content of the matrix `m` and the output of `layout.show`.
- Understand the parameters `widths` and `heights` in `layout`.
- Do you see how to obtain similar graphics with `mfrow` and `mfcol`?

Questions

1. Generate two samples `x <- rnorm(500)` and `y <- rf(500, 5, 5)` and display the scatter plot with the associated box plots (down horizontally for `x`, left vertically for `y`).

2. In the help page of `layout`, experiment and understand the example *Create a scatterplot with marginal histograms.*

3.3.2 Graphical Parameters

```
R > plot(LakeHuron, xlab="Year",
+ >   ylab="Level in feet",
+ >   main="Level of Lake Huron")
R > old.par <- par(bty="n",
+ >   col="red", bg="grey",
+ >   mar=c(2.5, 2.5, 2, 2),
+ >   mgp=c(1.5, 0.5, 0),
+ >   oma=c(0, 0, 0, 0),
+ >   cex.main=0.8,
+ >   cex.lab=0.7,
+ >   cex.axis=0.7)
R > plot(LakeHuron, xlab="Year",
+ >   ylab="Level in feet",
+ >   main="Level of Lake Huron")
R > par(old.par) # Reset settings
```

- The function `par` allows you to modify the graphical parameters. There are plenty of options, see `help(par)`.
- A good habit is to save the current parameters before doing any modification by keeping the object returned by `par`. When your graphics are done, call `par` with this object to reset all the parameters to their initial values. This is especially important when you tweak graphical parameters in the body of a function to avoid modifying the global settings.
- In the examples, we use various parameters related to framing. Use the help pages to understand their roles.

3.3.3 Axes and margins

```
R > x <- runif(50)
R > y <- runif(50)
R > plot(x, y)
R > plot(x, y, axes=FALSE)
R > axis(1)
R > axis(1, at=c(0.2,0.5,0.8), padj=1,
+ >   label=c("Low", "Average", "High"))
R > axis(2, lty=2, col=2)
R > mtext(c("A", "B", "C", "D", "E"),
+ >   side=2, at=seq(0.2, 1, by=0.2))
R > plot(x, y)
R > rug(x)
R > rug(y, side=4)
```

- The axes of a graph can be constructed retrospectively.
- The function `axis` adds an axis to the current graph. See `help(axis)` for details about the options.
- The function `mtext` permits to write text in the margins.
- The function `rug` is not related to axes or margins but stick to them.

Questions

1. Consider two vectors `v1 <- 1:10` and `v2 <- 100*sample(v1)`. What is the problem for displaying them together on the same graph?

2. Use the option `new` of the function `par` to represent `v1` and `v2` on the same graph with an axis on the right for `v1` and one one the left for `v2`.

3.3.4 Mathematical formulas

```
R > help(expression)
R > help(plotmath)
R > demo(plotmath)
R > plot(dnorm, -3, 3)
R > x <- seq(-3, 3, length=256)
R > lines(x, dnorm(x, 1, 1), col=2)
R > lines(x, dnorm(x, 0, 2), col=3)
R > title(expression(
+ >   over(1, sigma*sqrt(2*pi))
+ >   *e^{over(-(x-mu)^2, 2*sigma^2)}))
R > expr1 <- expression(mu==0~~sigma==1)
R > expr2 <- expression(mu==1~~sigma==1)
R > expr3 <- expression(mu==0~~sigma==2)
R > legend("topleft", c(expr1, expr2, expr3),
+ >   lty=1, col=1:3)
```

- Mathematical formulas can be written as text on the graphics in title, axes, legends, ...
- To write math, you need to use objects of type `expression`. The content of these objects has to be formatted according to a specific syntax derived from the famous TEX language of the awesome Donald Knuth.

3.3.5 Plotting networks

```
R > install.packages("igraph") # Take time
R > library(igraph)
R > g <- graph.ring(10)
R > plot(g)
R > plot(g, layout=layout.kamada.kawai,
+ >   vertex.color="green")
R > tkplot(g) # Interactive
R > g.lay <- layout.fruchterman.reingold(g, dim=3)
R > rglplot(g, layout=g.lay) # 3D
```

- The package `igraph` offers various functions to analyze networks. We only look at graphics ones in the examples.
- For more details, read the help pages about `igraph`, `plot.igraph` and `igraph.plotting`.

3.3.6 Geographical maps

```
R > install.packages("maps")
R > library(maps)
R > library(help=maps)
R > map("france", fill=TRUE, col=rainbow(10))
R > my.map <- map("france",
+ >   regions="haute-garonne", fill=TRUE,
+ >   col=grey(0.8))
R > title(my.map$names)
R > gps.coord <- matrix(c(1.443962, 43.604482),
+ >   nrow=1)
R > points(gps.coord, pch=16, cex=1.5)
R > text(gps.coord, labels="Toulouse", pos=3)
```

- There exist several packages which provide geographical data, `maps` is one of them.
- For geographical data analysis, you can look for information about packages `sp` and `GeoXp`.

3.4 Integrating R

3.4.1 R and OpenDocument

With the package `odfWeave`, you can insert R commands in an OpenDocument like **.odt* files of Libre Office Writer. This is a powerfull tool to produce elegant reports and properly integrate R outputs without modifying the initial file. The blocks of R commands are called chunks and are structured in the following way,

1. a line to introduce the chunk that contains `<<` followed by the chunk's name, some options, ... and ended with `>>=`,
2. your R commands,
3. a symbol `@` to end the chunk.

Let's take an example with a file *TestRaw.odt*,

```
This is an example of use of odfWeave.
<<chunk1, echo=TRUE>>=
x = rnorm(64)
summary(x)
@

It also works with graphics !
<<chunk2, echo=FALSE, fig=TRUE>>=
hist(rnorm(512), col='green')
@
```

Various options for the chunks are at your disposal in order to display or not the commands, to insert a figure, ... See `help(RweaveOdf)` for details. To produce the final document *Test.odt*, proceed as follows,

```
R > install.packages("odfWeave")
R > library(odfWeave)
R > odfWeave("TestRaw.odt", "Test.odt")
```

3.4.2 LaTeX

R fits perfectly with the powerful document markup language LaTeX. You can convert R objects to a formatted string to be directly included in your LaTeX document with the generic functions `toLatex` and `toBibtex` or with the package `xtable`,

```
R > toLatex(sessionInfo())
R > install.packages("xtable")
R > library(xtable)
R > xtable(data.frame(x=rnorm(10), y=runif(10)))
```

This relationship between R and LaTeX is widely used by packages like `Sweave` and `knitr` to produce high quality documents. LaTeX users and any interested readers should focus on these packages in order to see the quite limitless possibilities.