INTODUCTION TO R

This is only an introduction to the program `R`, in order to write and understand Monte Carlo simulations, in support to the studied theory. We recall the basic concepts of the language.

The easiest way to use `R` is in an interactive manner via the command line.

Under the opening message in the `R` Console is the
`>` ("greater than")
prompt. For the most part, statements in `R` are typed directly into the `R` Console window.
At the `>` prompt, you tell `R` what you want it to do. You give `R` a command and `R` does the work and gives the answer. If your command is too long to fit on a line or if you submit an incomplete command, a "**+**" is used for the continuation prompt.

One of the simplest (but very useful) ways to use `R` is to perform standard mathematical calculations. The `R` language includes the usual arithmetic operations: **`+,-,*,/,^`**. Some examples:

```
> 2+3
[1] 5
> 3/2
[1] 1.5
> 2^3 # this also can be written as 2**3
[1] 8
> 4^2-3*2 # this is simply 16 - 6
[1] 10
> (56-14)/6 – 4*7*10/(5^2-5) # this is more complicated
[1] -7
```

Other standard functions, that we will use, are for example: **`log(), gamma()`** Euler's gamma function, **`factorial()`** factorial function.

The assignment operator is "**<-**"; to be specific, this is composed of a **<** ("less than") and a **–** ("minus" or "dash") typed together. It is usually read as "gets. Alternatively, as of `R` version 1.4.0, you can use "**=**" as the assignment operator.
For example, in the following, the variable **`dieroll`** *gets* the value **`c(2,5,1,6,5,5,4,1)`**. Indeed, a useful command in `R` for entering small data sets is the **`c()`** function. This function *combines* terms together. For example, suppose the following represents eight tosses of a fair die:

`2 5 1 6 5 5 4 1`. To enter this into an `R` session, we type

```
> dieroll <- c(2,5,1,6,5,5,4,1)
> dieroll
[1] 2 5 1 6 5 5 4 1

>
```

Notice that we assigned the values to a variable called **`dieroll`**. `R` is case sensitive, so you could have another variable called **`DiEroLL`** and it would be distinct. The name of a variable can contain most combination of letters, numbers, and periods (.).

Each command you submit is stored in the History and the uparrow will navigate backwards along this history and the down arrow forwards. The left and right arrow keys move backwards and forwards along the command line.
You can add a comment to a command line by beginning it with the **#** character.
`R` ignores everything on an input line after a **#**.

All variables or “objects” created in R are stored in what’s called the *workspace*. To remove objects from the workspace (you’ll want to do this occasionally when your workspace gets too cluttered), use the **`rm()`** function. When exiting R, the software asks if you would like to save your workspace image. If you click yes, all objects (both new ones created in the current session and others from earlier sessions) will be available during your next session. If you click no, all new objects will be lost and the workspace will be restored to the last time the image was saved.

SCRIPT If you have a long series of commands that you would like to save for future use, you can write all of the lines of code in the script (text editor) and you can save the code in a file and execute them together selecting them and click on “Run” or using the **`source()`** function.
For example, we could type the following statements in a text editor (you *don’t* precede a line with a “>” in the editor):

```
x1 <- rnorm(500) # Simulate 500 standard normals
x2 <- rnorm(500) # ""
x3 <- rnorm(500) # ""
y1 <- x1 + x2
y2 <- x2 + x3
r <- cor(y1,y2)
```

If we save the file as **`corsim.R`** on the C: drive, we execute the script by typing

```
> source("C:/corsim.R")
> r
[1] 0.5085203
>
```

Note that not only was the object **`r`** was created, but so was **`x1`**, **`x2`**, **`x3`**, **`y1`**, and **`y2`**.

HELP There is text help available from within R using the function **`help()`** or the **`?`** character typed before a command. For example, suppose you would like to learn more about the function **`log()`** in R. The following two commands result in the same thing:

```
> help(log)
> ?log
```

Help can also be accessed from the menu on the R Console. This includes both the text help and help that you can access via a web browser. You can also perform a keyword search with the function **`apropos()`**. As an example, to find all functions in R that contain the string **`norm`**, type:

```
> apropos("norm")
[1] "dlnorm" "dnorm" "plnorm" "pnorm" "qlnorm"
[6] "qnorm" "qqnorm" "qqnorm.default" "rlnorm" "rnorm"
>
```

Note that we put the keyword in double quotes, but single quotes (`' '`) will also work.

MATRICES In order to understand how to generally use functions in R, let’s consider the function **`matrix()`**. This is a function that takes vectors and turns them into matrix objects. There are 4 arguments for this function, and they specify the entries and the size of the matrix object to be created. The argument **`byrow`** is set to be either **`TRUE`** or **`FALSE`** (or **`T`** or **`F`** – either are allowed for logicals) to specify how the values are filled in the matrix.
Often arguments for functions will have *default* values, and we see that all of the arguments in the **`matrix()`** function do. So, the call

```
> matrix()
```

will return a matrix that has one row, one column, with the single entry **`NA`** (missing or “not available”). However, the following is more interesting:

```
> a <- c(1,2,3,4,5,6,7,8)
> A <- matrix(a,nrow=2,ncol=4, byrow=FALSE) # a is different from A
> A
[,1] [,2] [,3] [,4]
```

```
[1,] 1 3 5 7
[2,] 2 4 6 8
>
```

Note that we could have left off the `byrow=FALSE` argument, since this is the default value. In addition, since there is a specified ordering to the arguments in the function, we also could have typed

```
> A <- matrix(a,2,4)
```

to get the same result. For the most part, however, it is best to include the argument names in a function call (especially when you aren't using the default values) so that you don't confuse yourself.

Vector Arithmetic

Vectors can be manipulated in a similar manner to scalars by using the same functions (however, one must be careful when adding or subtracting vectors of different lengths or some unexpected results may occur). Note that, by using these functions, the operations are done component by component. Some examples of such operations are:

```
> x <- c(1,2,3,4)
> y <- c(5,6,7,8)
> x*y
[1] 5 12 21 32
> y/x
[1] 5.000000 3.000000 2.333333 2.000000
> y-x
[1] 4 4 4 4
> x^y
[1] 1 64 2187 65536
```

Other useful functions that pertain to vectors include:

`length()` returns the number of entries in a vector
`sum()` calculates the arithmetic sum of all values in the vector
`prod()` calculates the product of all values in the vector
`cumsum()`, `cumprod()` cumulative sums and products

Some examples using these functions:

```
> s <- c(1,1,3,4,7,11)
> length(s)
[1] 6
> sum(s) # 1+1+3+4+7+11
[1] 27
> prod(s) # 1*1*3*4*7*11
[1] 924
> cumsum(s)
[1] 1 2 5 9 16 27
```

Matrix Operations

Among the many powerful features of `R` is its ability to perform matrix operations. As we have seen, you can create matrix objects from vectors of numbers using the `matrix()` command:

```
> a <- c(1,2,3,4,5,6,7,8,9,10)
> A <- matrix(a, nrow = 5, ncol = 2) # fill in by column
> A
[,1] [,2]
[1,] 1 6
[2,] 2 7
[3,] 3 8
[4,] 4 9
[5,] 5 10
> B <- matrix(a, nrow = 5, ncol = 2, byrow = TRUE) # fill in by row
> B
[,1] [,2]
```

```
[1,] 1 2
[2,] 3 4
[3,] 5 6
[4,] 7 8
[5,] 9 10
> C <- matrix(a, nrow = 2, ncol = 5, byrow = TRUE)
> C
[,1] [,2] [,3] [,4] [,5]
[1,] 1 2 3 4 5
[2,] 6 7 8 9 10
>
```

Matrix operations (multiplication, transpose, etc.) can easily be performed in `R` using a few simple functions like:

`dim()` dimension of the matrix (number of rows and columns)
`%*%` matrix multiplication
`t()` matrix transpose
`det()` determinant of a square matrix
`solve()` matrix inverse; also solves a system of linear equations
`eigen()` computes eigenvalues and eigenvectors

Moreover, there is the possibility to select components of vectors and matrices in the following way.

```
x<-c(2,5,9.5,-3)
> x[2] #select the second element of x
[1] 5
> x[c(2,4)] #select the elements in the positions 2 and 4
[1] 5 -3
> x[-c(1,3)] #exclude the elements in the positions 1 and 3
[1] 5 -3
> x[x>0] #select the positive elements
[1] 2.0 5.0 9.5
> x[!(x<=0)] #exclude the elements less or equal than 0
[1] 2.0 5.0 9.5
```

The parenthesis [] after the vector define which are the components to select.

For a matrix:

```
x<-matrix(1:10,ncol=5)
> x
[,1] [,2] [,3] [,4] [,5]
[1,] 1 3 5 7 9
[2,] 2 4 6 8 10
> x[,1]#select the first column
[1] 1 2
> x[2,]#select the second row
[1] 2 4 6 8 10
> x[2,3]#select the element in the position [2,3]
[1] 6
> x[,4:5]#selecl the columns 4 and 5
[,1] [,2]
[1,] 7 9
[2,] 8 10
> x[,-c(2,4)]#select the columns 1, 3 and 5
[,1] [,2] [,3]
[1,] 1 5 9
[2,] 2 6 10
```

Indexing of vectors and matrices

c() already seen

Sequences

Sometimes we will need to create a string of numerical values that have a regular pattern. Instead of typing the sequence out, we can define the pattern using some special operators and functions.

- The colon operator :

The colon operator creates a vector of numbers (between two specified numbers) that are one unit apart:

```
> 1:9
[1] 1 2 3 4 5 6 7 8 9
> 1.5:10 # you won't get to 10 here
[1] 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
> c(1.5:10,10) # we can attach it to the end this way
[1] 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5 10.0
> prod(1:8) # same as factorial(8)
[1] 40320
```

- The sequence function `seq()`

The sequence function can create a string of values with any increment you wish. You can either specify the *incremental value* or the desired *length* of the sequence:

```
> seq(1,5) # same as 1:5
[1] 1 2 3 4 5
> seq(1,5,by=.5) # increment by 0.5
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> seq(1,5,length=7) # figure out the increment for this length
[1] 1.00000 1.66667 2.33333 3.00000 3.66667 4.33333 5.00000
```

- The replicate function `rep()`

The replicate function can repeat a value or a sequence of values a specified number of times:

```
> rep(10,10) # repeat the value 10 ten times
[1] 10 10 10 10 10 10 10 10 10 10
> rep(c("A","B","C","D"),2) # repeat the string A,B,C,D twice
[1] "A" "B" "C" "D" "A" "B" "C" "D"
> matrix(rep(0,16),nrow=4) # a 4x4 matrix of zeroes
[,1] [,2] [,3] [,4]
[1,] 0 0 0 0
[2,] 0 0 0 0
[3,] 0 0 0 0
[4,] 0 0 0 0
>
```

Numerical Summaries

R includes a host of built in functions for computing sample statistics for both numerical (both continuous and discrete) and categorical data. For *numerical data*, these include, for example,

`mean()`	arithmetic mean
`median()`	sample median
`min()`, `max()`	smallest/largest values

Distribution Functions in R

R allows for the calculation of probabilities (including cumulative), the evaluation of probability density/mass functions, percentiles, and the generation of pseudo-random variables following a number of common distributions. For examples, we will use:

Distribution	R name	Additional arguments	Argument defaults
binomial	**`binom`**	**`size, prob`**	
uniform	**`unif`**	**`size, min, max`**	min=0, max=1

Prefix each R name given above with '**d**' for the density or mass function, '**p**' for the CDF, '**q**' for the percentile function (also called the quantile), and '**r**' for the generation of pseudorandom variables. The syntax has the following form – we use the wildcard ***rname*** to denote a distribution above:

```
> drname(x, ...) # the pdf/pmf at x (possibly a vector)
> prname(q, ...) # the CDF at q (possibly a vector)
> qrname(p, ...) # the pth (possibly a vector) percentile/quantile
> rrname(n, ...) # simulate n observations from this distribution
```

For example, with

```
> dbinom(3,size=10,prob=.25) # P(X=3) for X ~ Bin(n=10, p=.25)
```

we compute the probability to have 3 successes in 10 Bernoulli trials, with parameter of success equal to 0.25.

CONTROL-FLOW

The software R includes the usual control-flow statements, found in most programming languages. The principles are **if**, **if else** and **while**. They require the evaluation of a logical statement that can be expressed using logical operators, listed below:

Operator	**Meaning**
`==`	Equal to
`!=`	Not equal to
`<, <=`	Less than, less than or equal to
`>, >=`	Greater than, greater than or equal to
`&`	Logical AND
`|`	Logical OR

if, if else instructions

if (or **if else**) instruction performs a computation or an action (or different computations or actions) depending on whether a programmer-specified Boolean condition evaluates to **TRUE** or **FALSE**.

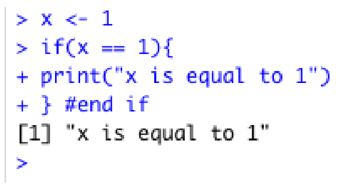
The usage in R is the following

```
if (cond) expr
if (cond) cons expr else alt expr
```

In **cond**, the conditions can be added, expressed in terms of a logical statement, which is of the type of length-one logical vector (it has not to be **NA**).

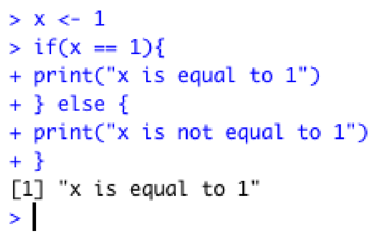
expr and **cons expr** are the resulting expression if the condition is **TRUE**, whereas **alt expr** is the alternative expression, used when the if-condition is not satisfied. Usually, they are expressions in a formal sense, and they can be either a simple expression or a compound expression, usually of the form **{expr1; expr2}**.

Example 1



```
> x <- 1
> if(x == 1){
+ print("x is equal to 1")
+ } #end if
[1] "x is equal to 1"
>
```

Example 2



```
> x <- 1
> if(x == 1){
+ print("x is equal to 1")
+ } else {
+ print("x is not equal to 1")
+ }
[1] "x is equal to 1"
>
```

`for` instruction

`for` instruction repeats a loop a certain number of times, counted by an index which increases its value of one unit at any iteration.

The usage in `R` is the following:

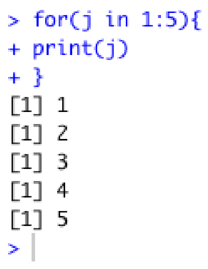
`for (var in seq) expr`

`var` is the syntactical name for a variable, the index of the for-loop.
`seq` is an expression evaluating to a vector, it contains the values of the index.

For example, if we write **`for (j in 1:5) expr`**, it means *"do expr for the different values of the index from 1 to 5"*. Note that it is not necessary to define previously the index-variable **`j`**.

As in **`for`**, **`if`** and **`if else`**, **`expr`** can be either a simple or a compound expression.

Example



```
> for(j in 1:5){
+ print(j)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
>
```

`while` loop

`while` loop is used to execute repeatedly a computation or an action until a programmer-specified Boolean condition is true.

The usage in `R` is the following:

`while (cond) expr`

The example computes the factorial of a number **`i`**.

Example

```
> factorial <- 1
> i <- 3
> while(i > 0) {
+ factorial = factorial*i
+ i <- i-1
+ }
> print(factorial)
[1] 6
```

OTHER USEFUL COMMANDS

`which` command

`which(x == a)` instruction returns a vector of the indices of **`x`** for which the comparison operation is **`TRUE`**, in this example the values of **`i`** for which **`x[i] = a`**.

Note that the argument of the function **`which`** must be a variable of logical type.

Example 1

```
> x <- c(1,2,3,1,1,5,6,1,1)
> which(x == 1)
[1] 1 4 5 8 9
```

Example 2

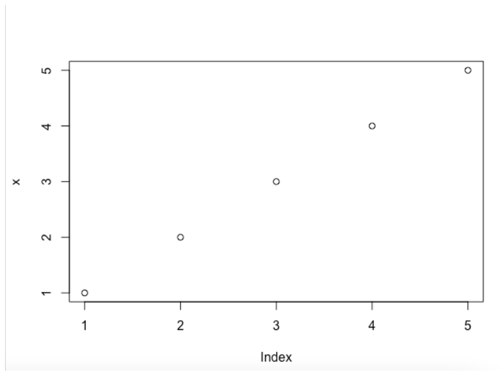
```
> x <- c(1,2,3,1,1,5,6,1,1)
> which(x <= 3)
[1] 1 2 3 4 5 8 9
```

`plot` command

The most common function used to graph anything in R is the **`plot( )`** function. This is a generic function that can be used for scatterplots, time-series plots, function graphs, etc.
If a single vector object is given to **`plot( )`**, the values are plotted on the y-axis against the row numbers or index.

Example 1

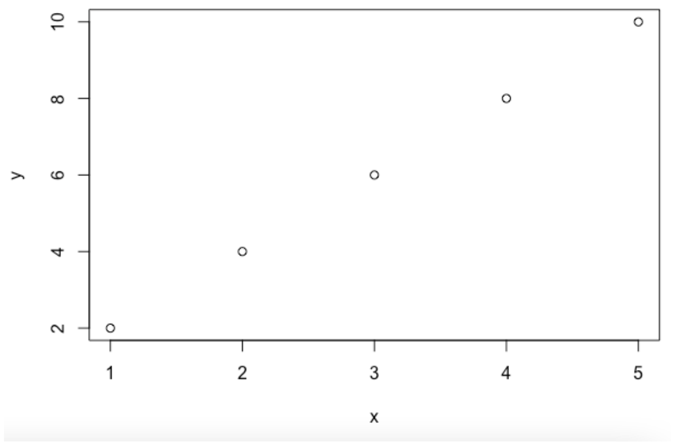
```
> x <- c(1,2,3,4,5)
> plot(x)
```



If two vector objects (of the same length) are given, a bivariate scatterplot is produced.

<u>*Example 2*</u>

```
> x <- c(1,2,3,4,5)
> y <- c(2,4,6,8,10)
> plot(x,y)
```



The default type for the **`plot`** is given by circular points and black in color. It is possible to change the plot type with the argument **`type`**, by using the following strigs.

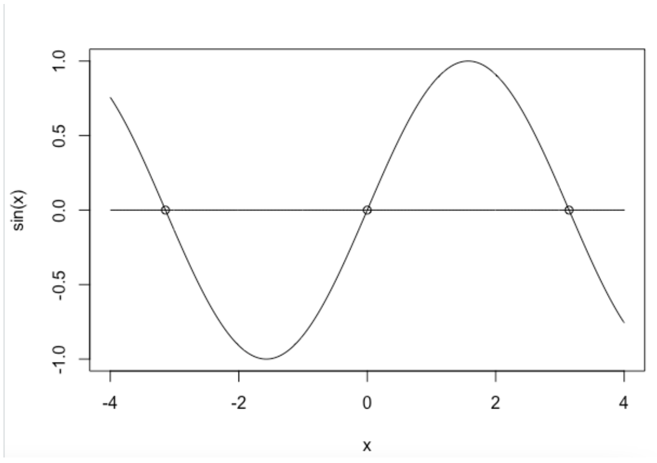
```
"p" – for points
"l" – for lines
"b" – for both points and lines
"c" – for empty points joined by lines
"o" – for overplotted points and lines
"s" and "S" – for stair steps
"h" – histogram-like vertical lines
"n" – does not produce any points or lines
```

Similarly, it is possible to define the color by using the argument **`col`**. The arguments **`lty`** and **`lwd`** allow the users to modify the lines type and the thickness.

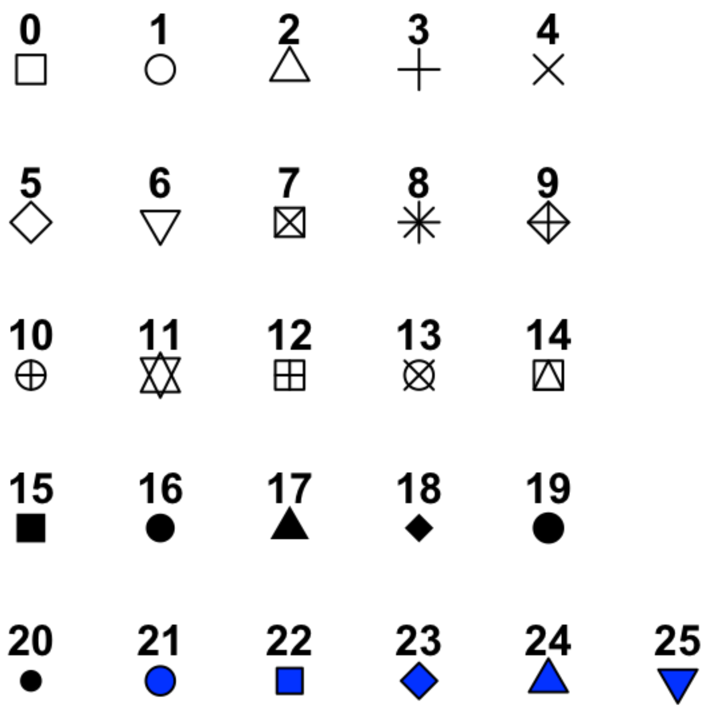
The command **`lines( )`** adds lines between coordinates, whereas **`points( )`** adds points at specified coordinates.

Example 3

```
> x <- seq(-4, 4, 0.01)
> a <- rep(0, times=length(x))
> b <- c(-pi,0,pi)
> c <- c(0,0,0)
> plot(x, sin(x), type="l")
> lines(x, a)
> points(b, c)
```



Different plotting symbols are available in `R`. The graphical argument used to specify point shapes is **`pch`**. The different points symbols commonly used in `R` are shown in the figure below



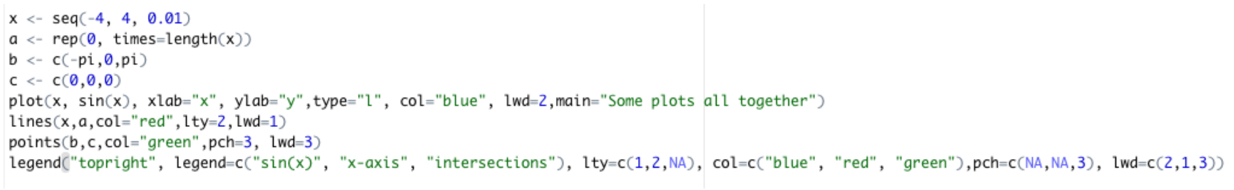
It is also possible to add personalized labels to the axes with **`xlab("...")`** and **`ylab("...")`**. Moreover, by means of the argument **`main("...")`** it is possible to insert a title on the top of the plot.

`legend` command

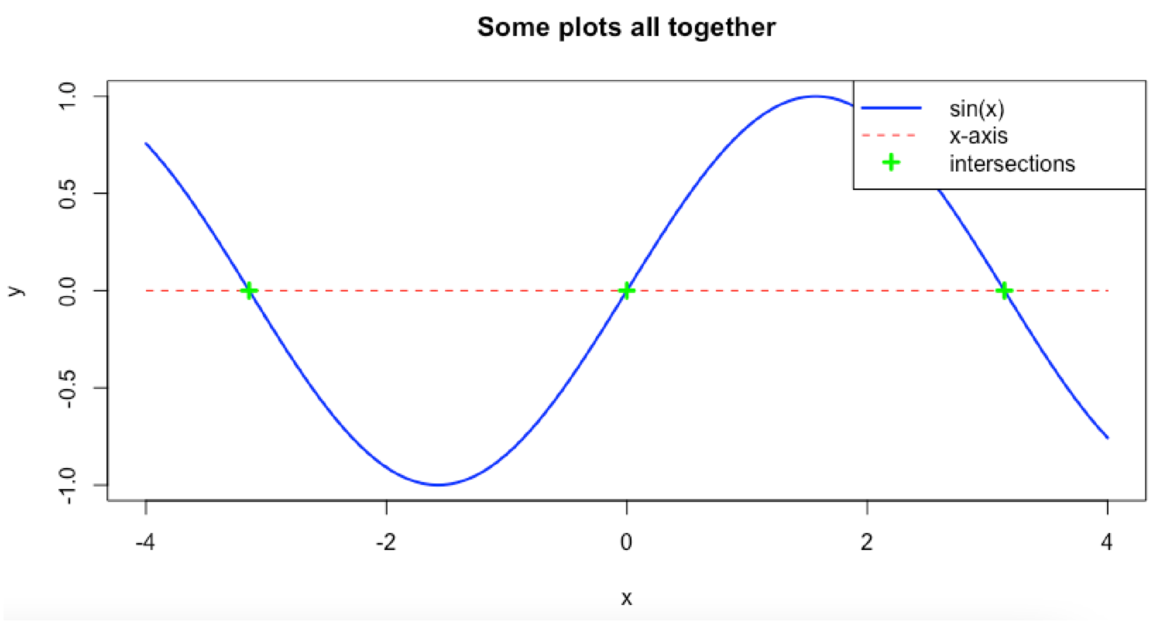
The legend function allows us to add a legend to a plot in base R. The syntax of the function is summarized below.

```
Legend(x, y       # Coordinates (x also accepts keywords, such as
                  "topright", "topleft", etc.)
       legend,  # Vector with the name of each group
       fill,    # Creates boxes in the legend with the specified colors
       col = par("col"), # Color of lines or symbols
       border = "black", # Fill box border color
       lty, lwd,         # Line type and width
       pch,              # Add pch symbols to legend lines or boxes
       bty = "o",        # Box type (bty= "n" removes box)
       bg = par ("bg"),   # Background color of the legend
       box.lwd = par("lwd"), # Legend box line width
       box.lty = par("lty"), # Legend box line type
       box.col = par("fg"),  # Legend box line color
       cex = 1,          # Legend size
       horiz = FALSE     # Horizontal (TRUE) or vertical (FALSE) legend
       title = NULL      # Legend title
      )
```

Example



```
x <- seq(-4, 4, 0.01)
a <- rep(0, times=length(x))
b <- c(-pi,0,pi)
c <- c(0,0,0)
plot(x, sin(x), xlab="x", ylab="y",type="l", col="blue", lwd=2,main="Some plots all together")
lines(x,a,col="red",lty=2,lwd=1)
points(b,c,col="green",pch=3, lwd=3)
legend("topright", legend=c("sin(x)", "x-axis", "intersections"), lty=c(1,2,NA), col=c("blue", "red", "green"),pch=c(NA,NA,3), lwd=c(2,1,3))
```



References:

- Paradis, E. (2005), *R for beginners*. https://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf

- Owen, W. J. (2010), *The R guide (version 2.5)*. https://cran.r-project.org/doc/contrib/Owen-TheRGuide.pdf

- R documentation