



First steps in R

Jorge Carneiro
<http://qobweb.igc.gulbenkian.pt>

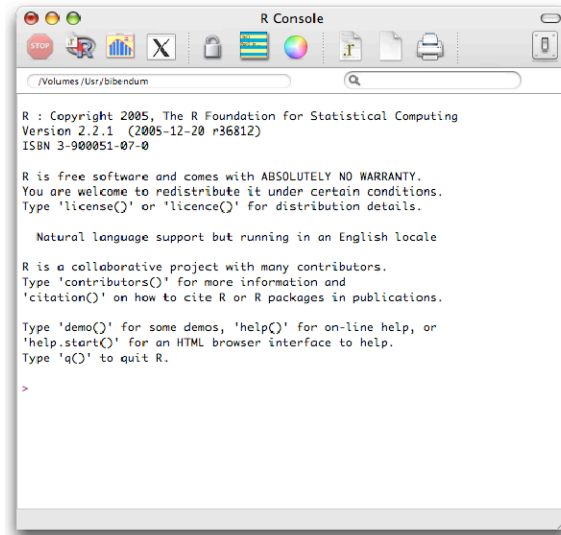
November, 2011

This tutorial represents the first practical session of the course 'Statistics and Quantitative Biology', held in November, in the context of the Gulbenkian PhD Programme in Integrative Biosciences (PIBS).

1. The very first steps

R is a command line application with powerful programming, statistical, and graphical capabilities. It is continuously being developed and improved, and it is free. Therefore it is highly recommended for the people initiating a research career.

R is distributed for all the basic platforms, including Windows, MacOSX, and Linux. On Windows and Mac you launch the application by clicking, while in linux you launch it by entering R on a console. Once you launch R you will get access to the R console where you can enter commands.



To quit R you use simply the command

```
> q()
```

R has a reasonable help system which you can access by entering:

```
> help()
```

To start with this you can read the help for the command `library()` by entering:

```
> help(library)
```

The command `library()` is used for loading and listing the packages available within R.

Additionally, there is online help that you can launch from the R console by entering:

```
> help.start()
```

This command will launch your default browser and load the online help system.

At anytime you can also try to find a match in the help for something. For example:

```
> help.search('transpose')
```

Finally, because it is always nice to learn by example you can use several use friendly demos that are available within R by entering the command:

```
> demo()
```

If you have read the help for the function `demo()` you know that you can also provide demo with arguments including the topic. Later on, in this tutorial, we will explore the graphics demo.

```
> demo(graphics)
```

On the console if you use the arrow keys 'up' and 'down' you can navigate through the history of commands and reenter them.

2. Data structures

2.1 Vectors

There are several ways of building a vector or ordered list. For example, the following commands can be used to build a vector `x` with entries 1 to 10.

```
> x <- 1:6
> assign("x", 1:6)
> x <- seq(1,6,by=1)
> x <- seq(length=6,from=1,by=1)
> x <- c(1,2,3,4,5,6)    # c stands for concatenate.
```

Enter `x` to confirm after each line that all these commands lead to the same result:

```
> x
[1] 1 2 3 4 5 6
```

Despite the giving the same result these commands have different potential (e.g. try to build a vector with arbitrary values using the first 4 commands !!).

You can reference or access the elements in the vector by their index by giving the vector followed by the indices in square brackets. You can specify individual entries or ranges as in the examples below:

```
> x<-c(2.4, 3.2, 4.8, 6.4, 9.6, 12.8)
> x[1]
[1] 2.4
> x[3:5]
[1] 4.8 6.4 9.6
> x[c(1,4)]
[1] 2.4 6.4
```

You can also specify elements by the value as in following examples:

```
> x[x>3]
[1] 3.2 4.8 6.4 9.6 12.8
> x[x>=3.2]
[1] 3.2 4.8 6.4 9.6 12.8
> x[x==3.2]
[1] 3.2
> x[x!=3.2]
[1] 2.4 4.8 6.4 9.6 12.8
```

Finally you can eliminate an element from the list by specifying a negative index:

```
> x[-2]
[1] 2.4 4.8 6.4 9.6 12.8
```

Notice that the two previous commands gave the same result.

A function that is very useful in manipulating vectors is [which](#). Read about it by entering [help\(which\)](#).

Try for example:

```
> which(x==2.4)
[1] 1
> which((x<=6.0)&(x>3.0))
[1] 2 3
```

By the way compare:

```
> x[which((x<=6.0)&(x>3.0))]
```

and:

```
> x[(x<=6.0)&(x>3.0)]
```

You can count the elements in a vector by the function:

```
> length(x)
[1] 6
> length(x[-2])
[1] 5
```

R has some built in arithmetic operations that work in an elementwise and can be nested, such as: +, -, *, /, sum, mean, ^, log, exp, sin, cos, tan, sqrt, abs, max, min, range, prod, cumsum.

Examine the help for all or some of these operations before you proceed.

```
> a<-6:1
> a
[1] 6 5 4 3 2 1
> b<-c(1,2,3,4,5,6)
> b
[1] 1 2 3 4 5 6
> a+b
[1] 7 7 7 7 7 7
> a-b
[1] 5 3 1 -1 -3 -5
```

```
> a-10
[1] -4 -5 -6 -7 -8 -9
```

By the way, examine a after all these operations by entering as usual:

```
> a
[1] 6 5 4 3 2 1
```

Compare with the result if you had entered:

```
> a<-a-10
> a
[1] -4 -5 -6 -7 -8 -9
```

If you want the operating to affect your vector you must assign the output of the operation the vector itself.

Try the other operations to get a feeling for what they do.

```
> sin(a)
```

Vectors and all other data structures or objects in R can be summarized using the command `summary()`. Try it:

```
> x<-c(2.4, 3.2, 4.8, 6.4, 9.6, 12.8)
> summary(x)
```

2.2. Matrices and arrays

Matrices or arrays are multi-dimensional data structures that can be indexed by two or more indices. You can build them in different ways. The most straightforward is by the command `matrix()` (as usual do `help(matrix)`)

```
> M<-matrix(1:20,4,5)
> M
      [,1] [,2] [,3] [,4] [,5]
[1,]   1    5    9   13   17
[2,]   2    6   10   14   18
[3,]   3    7   11   15   19
[4,]   4    8   12   16   20
```

The command `matrix` creates a matrix using the data vector `1:20`, and dimensions `4x5`, i.e. with 5 columns and 4 rows. Try substituting the data vector by any of the above mentioned ways of building vectors.

For example:

```
> c1<-1:5
```

```

> c2<-11:15
> c3<-21:25
> c4<-31:35
> M<-matrix(c(c1,c2,c3,c4),4,5)
> M
      [,1] [,2] [,3] [,4] [,5]
[1,]   1   5  14  23  32
[2,]   2  11  15  24  33
[3,]   3  12  21  25  34
[4,]   4  13  22  31  35

```

The elements in the matrix M are indexed much in the same way as those for vectors, but using two indices in square brackets. The following command return the element in first row and 4th column of M:

```

> M[1,4]
[1] 23

```

You can get a column (line) by just leaving the line index empty (columns) as in the examples:

```

> M[1,]
[1] 1 5 14 23 32
> M[,3]
[1] 14 15 21 22

```

You can get a submatrix by specifying ranges of indices as follows:

```

> M[1:3,1:3]
      [,1] [,2] [,3]
[1,]   1   5  14
[2,]   2  11  15
[3,]   3  12  21

```

Try summary of M as follows:

```

> summary(M)

```

2.3 Setting columns and rows names

The matrix M above has no column or row names. You can had them to the matrix as follows:

Try entering the following:

```

> M
> colnames(M)<-c("col1","col2","col3","col4","col5")
> M
> rownames(M)=c("row1","row2","row3","row4")
> M

```

Now you can access the columns, rows or elements by their name. Try entering:

```
> M["row2",]
> M[, "col3"]
> M["row2", "col3"]
```

2.4 Adding columns, rows or merging matrices

You can add a row to the matrix M by using the command `rbind()`.

```
> M<-rbind(M,c(-1,-2,-3,-4,-5))
```

The matrix M now contains an extra row which is unnamed. I

```
> M
      col1 col2 col3 col4 col5
row1    1    5    9   13   17
row2    2    6   10   14   18
row3    3    7   11   15   19
row4    4    8   12   16   20
      -1   -2   -3   -4   -5
```

To add or change the row name it is easy:

```
> rownames(M)[5]="row5"
```

What if you wish to remove a row or a column. You can do it using the negative index. To remove the fifth row you just added you can simply enter:

```
> M<-M[-5,]
```

The command `rbind()` and its sister command `cbind()` are useful also to merge matrices by columns or rows.

Let us create another matrix called M2 as above:

```
> M2<-matrix(26:50,5,5)
> M2
      [,1] [,2] [,3] [,4] [,5]
[1,]  26  31  36  41  46
[2,]  27  32  37  42  47
[3,]  28  33  38  43  48
[4,]  29  34  39  44  49
[5,]  30  35  40  45  50
```

To merge M and M2 by rows you can enter:

```
> M3 <-rbind(M,M2)
> M3
```

```

      col1 col2 col3 col4 col5
row1   1   5   9  13  17
row2   2   6  10  14  18
row3   3   7  11  15  19
row4   4   8  12  16  20
      26  31  36  41  46
      27  32  37  42  47
      28  33  38  43  48
      29  34  39  44  49
      30  35  40  45  50

```

Can you add the row names?

What about merging matrices M and M2 by columns ? To understand how `cbind()` works use it as follows:

```
> M4<-cbind(M,M2)
```

Error in `cbind(M, M2)` : number of rows of matrices must match (see arg 2)

You cannot merge the matrices by columns because they do not have the same number of rows. You can, however, choose a submatrix of M2 with 4 rows and merge it to M. For example:

```
> M4<-cbind(M,M2[1:4,])
```

```
> M4
```

```

      col1 col2 col3 col4 col5
row1   1   5   9  13  17 26 31 36 41 46
row2   2   6  10  14  18 27 32 37 42 47
row3   3   7  11  15  19 28 33 38 43 48
row4   4   8  12  16  20 29 34 39 44 49

```

Merging worked but the 5 last columns of M4 have no name. Can you give them names?

2.5 Lists

A list is a collection ordered objects or data structures that are called the lists' components. Components in a list can be of different types, and are referred by their name or index in double square brackets. Read the [help\(list\)](#).

```

> data <-list(measurements=matrix(rnorm(10),5,2),
+   treatmentgroup=factor(c("control","treatment" ))
+ )

```

```
> summary(data)
```

```

      Length Class Mode
measurements 10  -none- numeric
treatmentgroup 2   factor numeric

```

```
> data
```

```

$measurements
      [,1] [,2]

```



```
[1,] -0.7953982 0.7591176  
[2,] -0.3276192 -0.5352118  
[3,] -0.8854323 0.9472994  
[4,] 1.7953179 0.3135445  
[5,] -0.2481398 1.4826262
```

```
$treatmentgroup  
[1] control treatment  
Levels: control treatment
```

(Compare the values of numbers you got with those obtained by your neighbor. Surprised? `rnorm` generates a series of random numbers which are normally distributed with mean 0 and variance 1. Enter [help\(rnorm\)](#). We will get to work with this function later in this tutorial)

Try the following commands to see how to use a list and to reference its components and the elements within.

```
> Data[[1]]  
> Data[["measurements"]]  
> Data[[1]][5,1]  
> Data[["measurements"]][5,1]  
> Data[[2]][5,1]  
> Data[[2]][2]  
> Data[["treatment"]][2]
```

2.6 Data structures and their attributes

Data structure in R have attributes which can access through the function [attributes](#). Enter [help\(attributes\)](#).

Using the data structure [data](#) from previous section type the following:

```
> attributes(data)
```

R will return:

```
$names  
[1] "measurements" "treatmentgroup"
```

You can access associated to each attribute as, for example [data\\$measurements](#). Try entering:

```
> data$measurements  
> data$treatmentgroup
```

Each of these commands is in fact referring to a component of the data structure. You can access the elements within them by giving the corresponding indexes. Enter the following commands:

```
> data$treatmentgroup[2]
```

```
> data$measurements[1]
> data$measurements[,1]
> data$measurements[1,]
> data$measurements[1,2]
```

Do you understand the spirit of the thing ?

Now let us explore the data structures return by R itself. Consider the example of section 2.1.

```
> x<-c(2.4, 3.2, 4.8, 6.4, 9.6, 12.8)
```

Now let's store the summary of x in another variable say smry as follows:

```
> smry<-summary(x)
```

Now ask for the attributes of smry:

```
> attributes(smry)
```

R will return:

```
$names
[1] "Min." "1st Qu." "Median" "Mean" "3rd Qu." "Max."
```

```
$class
[1] "table"
```

You can now access for example the median or the mean by:

```
> smry["Mean"]
> smry["Median"]
```

You can also use these values to make computations. For example, you can compute the difference between the mean and the median of the data:

```
> smry["Mean"]-smry["Median"]
```

By the way, are you surprised that the difference between mean and median is so large ?

3. Flow control

R is a programming language which means you can implement algorithms. The key step is to understand what is the syntax of the commands involved in flow control of the program. To access this try entering any of the following commands:

```
> help('for')
> help('if')
```

As a trivial example suppose that you have a set of random numbers:

```
>x <-rnorm(15)
> x
[1] 0.6537516 0.1315439 0.0977684 1.1715284
[5] 0.2554547 -0.9632572 0.2261501 -0.2761696
[9] 2.6778739 0.4991180 -0.3916983 -0.4773608
[13] -0.3295837 0.4175589 -0.6393201
```

Suppose that you wish to build two vectors containing respectively the positive and negative elements.

Besides the obvious solution of resorting to the built-in capabilities of R, you would have to go through each element in the vector, compare it with zero, and, depending on the result of these comparison, copy it to one or another vector. The following code would do this:

```
> posv<-vector()
> negv<-vector()
> for (i in 1:length(x) ) {
+ if (x[i]>0) posv<-c(posv,x[i])
+ if (x[i]<=0) negv<-c(negv,x[i])
+ }
> posv
[1] 0.6537516 0.1315439 0.0977684 1.1715284
[5] 0.2554547 0.2261501 2.6778739 0.4991180
[9] 0.4175589
> negv
[1] -0.9632572 -0.2761696 -0.3916983 -0.4773608
[5] -0.3295837 -0.6393201
```

It is done! The code written above is not the most synthetic form of writing the conditions. It would be better to take advantage of expression

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

instead of two if' s has we have in the example. Can you rewrite the cycle accordingly ?

By the way, what would be most straightforward procedure using R capabilities ?

There is much more to programming in R than this simple flow control commands. However, addressing those more advanced topics is beyond the scope of this simple course.

3.4. Your first calculation in R

As an exercise try to calculate a complex formula. Consider a vector x containing n measurements. The sample variance is defined by:

$$\frac{1}{n-1} \sum_{i=1}^n \left(x_i - \frac{1}{n} \sum_{i=1}^n x_i \right)^2$$

To do this you will need to compute the sample mean, sum the square of the difference between each element and the mean, and finally divide by the number of measurements $n - 1$.

To simulate a series of 10 measurements type:

```
>x<-rnorm(10)
>x
[1] 1.9063343 -1.2332204 -0.9393411 0.3744104 -0.2157021 1.1368887
[7] -2.2505742 -0.3406168 0.4532608 0.6785361
```

To defined n you can simply enter:

```
> n<-length(x)
[1] 10
```

How to compute the arithmetic mean ? Either using the built-in function `mean()` or by summing over all the n elements of the vector and divide by n.

For example:

```
> m<-0
> for (i in 1:n) { m<-m+x[i]}
> m
[1] -0.4300241
> m<-m/n
> m
[1] -0.04300241
```

Using the function `mean` is much easier and (obviously) gives the same result:

```
> mean(x)
[1] -0.04300241
```

Back to the sum of the difference to the mean squared it should be easy now:

```
> ss<-0
> for (i in 1:n) { s<-(x[i]-m)^2
> ss<-ss+s
> }
> ss
[1] 13.345
```

or simply:

```
> sum((x-mean(x))^2)
[1] 13.345
```

Finally divide by $n-1$:

```
> ss/(n-1)
[1] 1.482778
```

The whole procedure in a stepwise manner without using the built-in functions of R was:

```
> n<-length(x)
> m<-0
> for (i in 1:n) { m<-m+x[i]}
> ss<-0
> for (i in 1:n) { s<-(x[i]-m)^2
> ss<-ss+s
> }
> ss/(n-1)
[1] 1.482778
```

The same result is obtained with more 'economical' expression:

```
> sum((x-mean(x))^2)/(length(x)-1)
```

or simply using:

```
> var(x)
```

4. Reading and writing data to files

4.1. Importing data spreadsheet type text files

The spreadsheet text format come in different flavors, in which the data are presented as a rectangular grid, possibly with row and column labels, and separated by special characters (such as space, tabs, or commas).

The function `read.table` is the best way of reading data from a text file. The first argument is the name of the file and it should be the absolute path unless you set the working directory to that where your file is.

```
> Data<-read.table("CitationsExpModels.txt")
> Data
```

	C_elegans	D_melanogaster	H_sapiens
Anatomy	3725	7814	1644813
Physiology	10165	22511	3624921
Biochemistry	537	1147	157250
Genetics	7998	18798	825834
Development	61	98	1359
Microbiology	149	313	317993

Pathology	295	356	1138227
Pharmacology	1698	3964	2080405
Behavior	581	1721	655900
Ecology	12	51	6913

Try a few things like:

```
> Data['Anatomy','C_elegans']
```

```
[1] 3725
```

```
> Data['H_sapiens']
```

```

H_sapiens
Anatomy      1644813
Physiology   3624921
Biochemistry 157250
Genetics     825834
Development  1359
Microbiology 317993
Pathology    1138227
Pharmacology 2080405
Behavior     655900
Ecology      6913

```

```
> Data[[1]]
```

```
[1] 3725 10165 537 7998 61 149 295 1698 581 12
```

```
> summary(Data[[1]])
```

```

Min. 1st Qu. Median Mean 3rd Qu. Max.
12.0 185.5 559.0 2522.0 3218.0 10170.0

```

```
> summary(Data[[2]])
```

```

Min. 1st Qu. Median Mean 3rd Qu. Max.
51.0 323.8 1434.0 5677.0 6852.0 22510.0

```

```
> summary(Data[[3]])
```

```

Min. 1st Qu. Median Mean 3rd Qu. Max.
1359 197400 740900 1045000 1518000 3625000

```

4.2. Export data spreadsheet type text files

Try the function `write.table` using the table above. For example:

```
> write.table(Data,"name_of_file")
```

Open the file `"name_of_file"` and see with another software.

Try also:

```
> Data2<-read.table("name_of_file")
```

```
> Data2
```

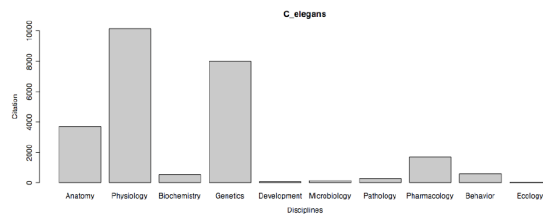
5. Graphics

The first step to understand how to make graphical representation of data is to run the command:

```
> demo(graphics)
```

After playing with the demos try the following commands:

```
> barplot(Data[[1]],names.arg =rownames(Data))
> barplot(Data[[1]],names.arg =rownames(Data), xlab="Disciplines")
> barplot(Data[[1]],names.arg =rownames(Data), xlab="Disciplines", ylab="Citation")
> barplot(Data[[1]],names.arg =rownames(Data), xlab="Disciplines", ylab="Citation", main=colnames(Data)[1])
```



```
> pie(Data[[3]], labels=rownames(Data))
> pie(Data[[2]], labels=rownames(Data))
> pie(Data[[1]], labels=rownames(Data))

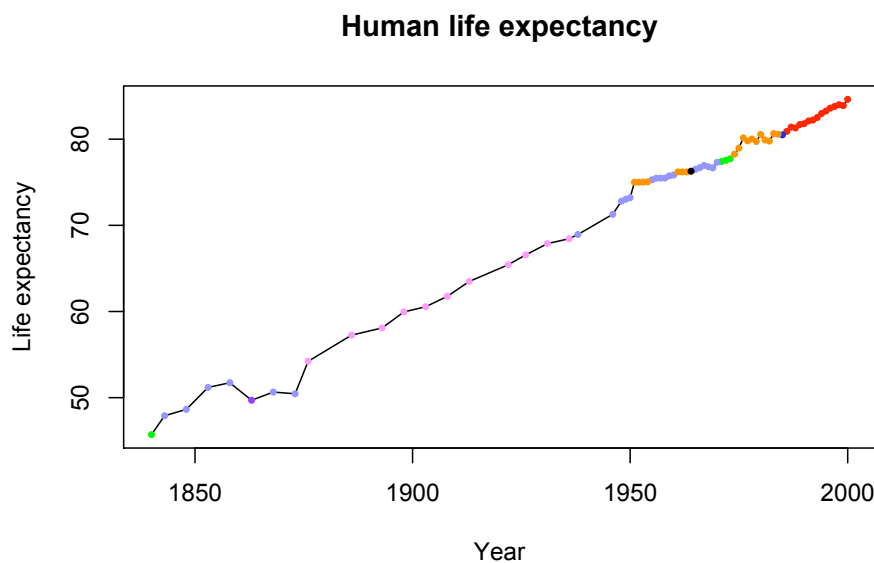
> pie(Data[[1]], labels=rownames(Data),main=colnames(Data)[1], col=rainbow(length(Data[[1]])))

> plot(Data[[1]],Data[[2]])
> plot(Data[[1]],Data[[2]], xlab=colnames(Data)[1])
> plot(Data[[1]],Data[[2]], xlab=colnames(Data)[1], ylab=colnames(Data)[2])
> plot(log(Data[[1]]),log(Data[[2]]), xlab=colnames(Data)[1], ylab=colnames(Data)[2])
```

The next step should be browsing the library graphics to see all the potentiality of R by entering `help(graphics)` and following the links. Particularly useful for the remaining of the course will be the plotting commands `mosaicplot` and `boxplot`:

```
>help(mosaicplot)
>help(boxplot)
```

As a final exercise on graphs read the data set in file “HumanLifeExpectancy.txt” and reproduce the following graph:



There is no legend in the graph. Can you had one ?

6. Creating (pseudo)random data

In this course we are going to resort often to simulated data and to random resampling of data. We have already used this R capability when generating data with the function `rnorm`. R has already built-in procedures to generate random data following many different canonical distributions. To name a few: `rbinom`, `rbeta`, `rchisq`, `rpois`, and `rexp`. As usual evoke the help for each of these functions.

Let us practice this a bit:

```
>x1 <-rnorm(10, 1.6,0.1)
>x2 <-rnorm(100, 1.6,0.1)
>x3 <-rnorm(1000, 1.6,0.1)
>x4 <-rnorm(10000, 1.6,0.1)
```

Make histograms of each vector and calculate mean and variance.

```
> n<-vector()
> for (i in 1:100) { n<-c(n,5*i)}

> v<-vector()
> m<-vector()
> for (i in 1:100) {x<-rnorm(n[i],1.6,0.1)
+ m<-c(m,mean(x))
+ v<-c(v,var(x))
+ }

> plot(n,m)
```



```
> plot(n,v)
```

Let us now try to compute histogram of the position of a sphere that we let fall into a grid that forces unbiased binary bifurcation. We will use the function `sample` (as usual do `help(sample)`).

```
> x<-c(-1,1)
> sp<-sample(x,10,replace=TRUE)
> sp
[1] 1 1 -1 1 -1 1 1 -1 1 1

> pos<-vector()
> for (i in 2:length(sp)) {
+ pos<-c(pos,sum(sp[1:i]))
+ }
> pos
[1] 2 1 2 1 2 3 2 3 4

> pv<-vector()
> for (z in 1:1000) {
+ sp<-sample(x,10,replace=TRUE)
+ pos<-vector()
+ for (i in 2:length(sp)) { pos<-c(pos,sum(sp[1:i])) }
+ pv<-c(pv,pos[length(pos)])
+ }
> plot(pv)
> hist(pv)
```