Introduction to R
Programming and Statistics Environment

Christoforos Nikolaou

https://github.com/christoforos-nikolaou/MolBioMedClass

What is R

R is a language and environment for statistical computing and graphics. It is a GNU project 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 is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes:

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or on hardcopy, and
- a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

Why use R

R is a versatile, dynamic and fast way to

- Import data, especially big chunks of data in an programming environment
- Clean data, by removing missing or corrupted values and by easily transforming data to desirable formats
- Summarize data, getting descriptive statistics with straightforward commands
- Visualizing data, with elegant plots
- Modeling data, by performing state-of-the-art analyses

Getting and Starting R

Getting R

Is very easy. In Linux systems just type:

```
sudo apt-get install r-base-core
```

in your terminal.

In both Linux and Windows you can try R-Studio, a GUI-style R with split windows holding history, plots and file separately for advanced control (at a certain expense of speed and memory)

Get it here: http://www.rstudio.com/

Running R

Is even easier. In Linux you simply type:

R

in your terminal and the environment loads automatically. You can now start working. R-Studio starts either via the command line or by clicking the approprate button on your programs folder.

Components of R

Data types:

The way R “sees” data. It is very important to know what type is your data and how you can switch between data types

Operators:

Arithmetical operators, simple commands etc.

Functions:

R's power. They can be from simple to very complex series of calculations that perform mathematical and statistical analysis and create plots. It is very important to know which data type you can use them with.

Commands/Programming:

More advanced ways to create and set up you own analyses (we want be talking about it).

Data Classes

Data types are important in R. Most, if not all of its functions, are to be executed on specific type(s) of data. It is therefore crucial that you make sure you are using the right one.

Data types (pieces of data) belong to one of the following five **classes**:

- character (e.g. "Christoforos", "A33")
- numeric (real numbers) (e.g. 3.14)
- integer (e.g. 6)
- complex (e.g. -1+4i)
- logical (True/False or in some cases T/F)

Data Types

Data objects are organized in structures which we call **data types** and which may be:

- simple variables: No-dimension singular values
- Vectors: One-dimensional enumerated arrays of the same class of data
- Matrices: Two-dimensional matrices of numerical data
- Data frames: Two-dimensional tables that can hold different classes of data in each column
- Factors: Special types of vectors that handle categorical data
- Lists: Data-frames that don't necessarily need have fixed dimensions

Single Variables

Simple variables are the most basic data type:

```
1
```

```
"Me"
```

```
0.98
```

```
TRUE
```

We can create them with a very simple command like:

```
x<-1
```

```
name<-"Me"
```

Which is called "assignment". The "<-" is the assignment operator and works like this:

```
Name_of_variable <-  value
```

Vectors

Vectors may be created with the simple function "concatanate" c or with the use of the vector function

```
x <- c(1,2,3)
```

```
y <- c("me","you","him")
```

The vector function is to be used mostly for initialization purposes

```
z <- vector("numeric", length=20)
```

This creates a vector of 20 "0" values.

How can we access parts of data of a vector? Simply by asking for the “number” of the data in one dimension

`z[1]` # will print on the screen the first element of z

`Z[5:10]` # will print on the screen the elements from 5th to 10th

`z[c(1,5,7)]` # what does this do?

Matrices

Matrices are introduced with the matrix function. As with vectors, matrices need to be assigned with dimension specifications. Matrices need two dimensions as they are two-dimensional.

```
m <- matrix(0, nrow=2, ncol=5)
```

creates a 2x5 matrix of zeros

The dimensions of a matrix can be retrieved with the dim function

```
dim(m)
```

R fills matrices by completing the columns, starting from the upper left part (element[1,1]). So if you wanted to fill m with the first 10 numbers that would be done by:

```
m <- matrix(1:10, nrow=2, ncol=5)
```

If we now wanted to see what m looks like, we would simply type:

```
m
```

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

Matrices

We can also create a matrix from a vector by adding a dimension attribute. The dimensions are in this case a vector themselves

```
x <- 1:10 # a vector of the numbers 1 to 10

dim(x) <- c(2,5) # dimensions read as number of rows, number of colums

m #will print m on screen like this:

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

Two very useful functions for matrix manipulation allow us to add, join rows and columns to an existing matrix, or create matrices by joining vectors.

rbind joins vectors by rows and cbind does the same by columns

```
x <- 1:10

y <- 11:20

z <- rbind(x,y) # join x and y by treating them as rows of a matrix called z

z # show z
```

Matrices

How can we access a matrix in a way similar to the one we did for the vectors? Remember matrices are two-dimensional, which means we need two values to define each element.

By definition we use the first value to account for the row and the second to account for the column. Which means that:

`m[1,2]` # returns the element that lies in the 1st row and the 2nd column

Use of the indices can be very elaborate. For instance, think what does this return?

```
m[c(1,6,9), 2:12]
```

Data Frames

Data frames are one of the most common and versatile data type in R. They are tabular lists and so can contain elements of different classes, but they are also matrix-like in the sense that all vectors in the list should be of the same size (length). Data frames are mostly read-in in R with specific commands (see Reading Data).

When not reading data frames from a file already stored in the computer, we can declare them with commands like:

```
x<- data.frame(a=1:4, b=c("Me","You","Him","Her"))

x

  a   b
1 1  Me
2 2 You
3 3 Him
4 4 Her
```

Data Frames

Notice that the columns have names ("a" and "b") which were given in the declaration of the variable. Also notice that rows are numbered from 1 to 4. These can be recalled with the row.names() function

```
row.names(x)
```

```
[1] "1" "2" "3" "4"
```

The size of the data frame can be given either with dim() or by calling the data frame-specific functions nrow() and ncol()

```
dim(x) # [1] 4 2
```

```
nrow(x) # 4
```

```
ncol(x) # 2
```

Data frames are the most commonly used data type, especially when handling external data (files from your computer). We will see more of that later on.

Factors

Imagine your data are based on some categorical, non-numeric variable such as "good", "bad", or "diseased","healthy" etc. In this case you will need a data type that deals with non-numeric data.

Factors are special types of vectors that handle categorical (non-numerical) data. Their main (and outmost) difference from vectors is that **they are labeled instead of ordered**. This means that a factor has "names" such as "Me", "You" and "Him" instead of numbers such as 1, 2, 3 designated to its elements. In this sense, they are much more useful when trying to address different subsets of the data, a very important aspect of analysis that is called Subsetting.

Inserting a factor is as easy as:

```
fac<-factor(c("me", "me", "you", "me", "him", "you"))
```

Notice that we actually call a function called factor() upon a vector, thus we say we "factorize" a vector. fac now holds the names of the variables "me", "you" and "him" but it does so in specific positions.

The different categorical values held can be visualized with the use of the levels() function

```
levels(fac)
```

```
[1] "him" "me"  "you"
```

```
      3   2   1
```

Factors

Below the "numerized" factor, R also returns the attributed levels, a sort of legend that tells you which number corresponds to which. Not all factors you 'll be dealing with will be that small though and so it would be easy to have a summary of the levels representation in the factor. This is returned with the use of the table() function.

```
table(fac)

fac

him  me you

  1   3   2
```

which returns the levels ordered by attribute number and the corresponding number of elements below it. This tells us that in our fac factor we had 1 instance of "him", three instances of "me" and two of "you".

Lists

Lists are data frames that do not have to follow the restriction of equal vector size. In this sense you may see them as data "blobs" that can hold simple variables or vectors of any type or size. For the moment it would be useful to know how to introduce one. Lets do it step by step, by creating three vectors first:

```
vec <- c(1,5,7)

mec <- 1:10

dec <- c("TRUE","TRUE","FALSE","FALSE","FALSE")
```

Now let's put them all in a list with that order

```
l<-list(vec,mec,dec)
```

l now contains vec, mec and dec in this order. Which means that if call back the first element of l (by asking for l[1]) we will be getting the complete vector vec

```
l[1]

[[1]]

[1] 1 5 7
```

Coercion of data types

Data types containing mixed objects are to be treated with extreme caution. This is because R coerces data

```
y <- c(1.7, "a") # y is now character
```

```
y <- c(TRUE, 2) # y is now numeric
```

We can find out the the class of a data type by typing

```
x <- c(1,2,"TRUE")
```

```
class(x)
```

and coerce the data type to the one we desire with the as."" function

```
as.logical(x)
```

```
[1]   NA   NA TRUE
```

Caution: when the coersion makes no sense, R returns the "NA" variable.

Be prepared to see this a lot if you are not careful with your data assignments.

What class is my data: str()

What if we wanted to know what is the data type of each of the object in l? In this case we may use the str() function

```
str(l)

List of 3

 $ : num [1:3] 1 5 7

 $ : int [1:10] 1 2 3 4 5 6 7 8 9 10

 $ : chr [1:5] "TRUE" "TRUE" "FALSE" "FALSE" ...
```

str(l) returns a much more detailed output that contains the data class of l (List), the number of objects in it (of 3) and the data type of each of the objects (num, int, character) alongside the first instances in each one. Notice how the last object of l has been assigned the type "character" (chr). We can easily coerce it back to logical and put it in the list with

```
l<-list(vec,mec,as.logical(dec))

str(l)

List of 3

 $ : num [1:3] 1 5 7

 $ : int [1:10] 1 2 3 4 5 6 7 8 9 10

 $ : logi [1:5] TRUE TRUE FALSE FALSE FALSE
```

Subsetting data

What if we wanted to know what is the data type of each of the object in l? In this case we may use the str() function

```
str(l)

List of 3

 $ : num [1:3] 1 5 7

 $ : int [1:10] 1 2 3 4 5 6 7 8 9 10

 $ : chr [1:5] "TRUE" "TRUE" "FALSE" "FALSE" ...
```

str(l) returns a much more detailed output that contains the data class of l (List), the number of objects in it (of 3) and the data type of each of the objects (num, int, character) alongside the first instances in each one. Notice how the last object of l has been assigned the type "character" (chr). We can easily coerce it back to logical and put it in the list with

```
l<-list(vec,mec,as.logical(dec))

str(l)

List of 3

 $ : num [1:3] 1 5 7

 $ : int [1:10] 1 2 3 4 5 6 7 8 9 10

 $ : logi [1:5] TRUE TRUE FALSE FALSE FALSE
```

Structural Subsetting

Makes use of the [], [[]] and $ operators, which with the clever combination of commas can provide absolute precision on the choise of data with only a few characters coding.

[] may be used on any vector, factor, matrix or dataframe to subset it in one or two dimensions. In the case of vectors and factors there is only one dimension. Therefore, if we want the nth element of a vector v we simply put n within brackets

```
x<-v[6]
```

x now holds the sixth value of the vector v. R enumerates all data types starting from 1 so 6 will actually return the 6th value.

```
x<-v[6:8]
```

will get a "slice" of v and store it in x which now becomes a vector itself carrying the 6th,7th and 8th elements of v. Remember how the ":" operator is used for ordered integers.

Structural Subsetting

Now what if we wanted some compartmentalized subsetting that does not follow a certain order

```
x<-v[c(1, 6:8, 11:15, 18, 20)]
```

This intricate subsetting allows us to get the 1st, then the 6th-8th, then the 11th-15th, then the 18th and then the 20th elements of v and store them in a vector called x. Notice how the subsetting indices (the numbers in the parentheses) are a vector in themselves and thus they are introduced with the concatanate function c(). We could have greater control in this subsetting if we split the process in two

```
ind<-c(1, 6:8, 11:15, 18, 20)

x<-v[ind]
```

This first creates a vector called ind that carries the indices (the numbers of elements we want to obtain from v) and then passes it to v with [] to perform the subsetting.

Structural Subsetting

But what about matrices and dataframes? Here there are two dimensions on which we can subset (rows and columns). R uses the same operator [] but allows for two values separated by comma to provide information of rows and columns (in this order). Suppose we need to keep only the first and the third line from a matrix m. This is done with:

```
mm <-  m[,c(1,3)]
```

In perfect symmetry subsetting on rows 10 through 20 would be performed with:

```
mm <- m[10:20,]
```

If m was a MxN matrix, then mm is a Mx2 in the first case and 10xN in the second. In the case we subset on both rows and columns

```
mm <- m[c(1:5,8), c(10,11, 12:14)]
```

mm is now a 6x5 matrix.

Structural Subsetting

In the case of data frames with named columns we can also use the $ operator to subset columns. Take the built-in R data frame called mtcars simply by typing

```
mtcars
```

This small dataframe contains makes of cars alongside their constructor specifications. In order to choose one specific specification simply ask for the name of the data frame followed by $ and the name of the column. For instance

```
mtcars$cyl
```

will return the vector containing the cyl column. This is identical to calling mtcars[,2] asking thus for the second column of the data frame (not including the names of the cars).

Logical Subsetting

Makes use of logical operators ("&", "!", "|", see more on that in Control Structures)

Logical operators stand for AND (="&"), OR (="|") and NOT ("!"). While the first two are more complex as "joining" operators and we will see more of them when we discuss control structures, NOT "!" can stand on its own as it signifies the negation of a statement. In this sense

```
!is.na(x)
```

returns logical values (TRUE or FALSE) depending on whether any value in x is NOT "NA". In this sense, !is.na() is the mirror-image of is.na(). Lets use this in a subset

```
y<- x[!is.na(x)]
```

y is now a vector with all the elements of x that are NOT "NA".

Numerical Subsetting

What if we wanted to get values that are greater than 2. We could try

```
x[(x>2)]
```

and we would get

```
[1] "3" "4" NA  NA  NA  NA

Warning message:

In Ops.factor(left, right) : > not meaningful for factors
```

What happened here?

We got the two numerical values that fulfill our restriction (>2) but we got four NA values at the end and an error message that said our operation is not meaningful for factors.

What R is trying to tell us is that a comparison >2 is not possible for categorical data like "Me" or "You". In this sense the operation "Me">2 returns nothing (NA). R prints this at the end of the vector but is kind enough to point it out to us.

Numerical Subsetting

Numerical subsetting is not always numerical. It can be categorical as well with the use of the "==" and the "!=" operators. If we try

```
x[(x!="Me")]
```

```
[1] "1"   "2"   "3"   "4"   "You" "Him" "Her"
```

we get back all the elements of x that are not equal to "Me" regardless if they are numbers or characters. This is because R performs coercion of variables wherever this is possible before conducting the subsetting.

One very handy function for subsetting is **which()**. It can be used for both logical and numerical subsetting to produce a subset of indices fulfilling certain conditions.

```
which(x>=1)->ind
```

```
x[ind]->new_x
```

R operators

We have already seen the basic “assignment” operator “->”.

Other operators in R are the ones that carry out basic arithmetic operations:

```
x <- 2

y <- x+2 # addition/subtraction

z <- x*y # multiplication

d <- z/4 # division

d**1.5 # power (same as d^1.5)
```

Precedence is based on the normal mathematical rules of precedence (**, */, +-) so you must always use brackets when coding a more complicated formula. Brackets are used in a nested manner like:

```
p <- 2*((x-y)**2)-3.14
```

R operators

Logical operators. These are the ones with which we control statements of a logical type such as

== : equation (e.g. a==b checks a condition of equality between a and b)

!= : not equal (checks non-equality)

>, <, <=, >= : greater/smaller than, greater or equal/smaller or equal

| : disjuntion (OR) (eg. X | Y means “X or Y”)

& : AND (eg. X & Y means “X AND Y”)

Apart from those, other operators that we need to now are:

(...) #parentheses create the space for the use of functions

[..] #brackets are used to provide indexes for vectors, factors, matrices etc

: #is used to create series of integer values (e.g. 1:5 is 1,2,3,4 and 5)

R Functions

Functions are the power of R. They are predefined commands that act on a set of variables to yield results that can range from simple calculations (e.g sums, mean values) to complex data structures (e.g. clusterings, intricate plots etc).

The basic structure of functions is:

```
function(var1, var2 …., parameter1=value, parameter2=value....)
```

Where the variables and the parameters are separated with commas.

Simple functions are self explanatory and we can run the directly. E.g.

```
x<-c(1,2,3,5,8,13,21,34)

m<-mean(x)
```

m will now carry the mean value of the numerical vector x.

Important: Different functions act on different data types. ALWAYS know what the function needs!!!

Numeric Functions

Numeric functions include functions used to performed more advanced mathematical operations.

Some of the most important are:

abs(x) : absolute value

sqrt(x) : square root

ceiling(x), floor(x), trunc(x), round(x, digits=n), signif(x, digits=n)

All of these functions treat real numbers in terms of rounding (that is ommission of decimal points). ceiling(x) rounds up the real number x to the closest integer that is greater than x (>x) while floor(x) does the opposite, rounding x to the closest integer that is smaller than x (<x). trunc(x) truncates all decimal points rounding the number to the closest integer in the same way floor() does it. round() and signif() both take a number of digits as an additional argument but slightly differ in that round(x, digits=n) does rounding in a way that keeps n decimal points while signif(x, digits=n) rounds to n total digits (not including the comma).

cos(x), sin(x), tan(x), acos(x), cosh(x), acosh(x)

log(x) : natural logarithm, **log10(x)** : common logarithm

Character Functions

Character handling is not one of R's major strong points and you can always work around data manipulation in character strings outside R with shell scripting or other scripting languages. Still, R provides a number of functions we can use to handle strings when we need to stay within the environment.

substr(x, start=n, stop=m)

The substr() function act on a string x, from which it returns a substring starting from n and running through m.

grep(pattern, x , ignore.case=FALSE, fixed=FALSE)

Pattern matching in R is performed with grep() that searches for pattern in x. Notice that the functions returns the matching indices, that is the result of the matching is the subset of elements of the vector x that match the pattern.

A substitution function sub() is similar to grep but with the addition of a replacement string

sub(pattern, replacement, x, ignore.case=F, fixed=F)

Character Functions

Splitting of a string is performed with strsplit()

strsplit(x, split)

where split is the character(s) at which splitting takes place. strsplit() returns a character vector. As in the previous functions, fixed=T allows split to be a regular expression. For instance:

```
x<-"abc.d.eef.g"

strsplit(x,".", fixed=T)->d

d

[[1]]

[1] "abc" "d"   "eef" "g"
```

The paste() function joins strings in a concatenation using sep as a separator

```
paste("x",1:3,sep="")
```

returns c("x1","x2" "x3") while

```
paste("x",1:3,sep="M")
```

returns c("xM1","xM2" "xM3").

Transforming strings to upper or lower case is explicitly done with **toupper(x)** and **tolower(x)**.

Other general functions

Suppose you need to generate a sequence of numbers with a fixed interval. Remember that this can be done with the ":" operator only for interval=1 but if we want another step we may use

seq(from , to, by)

If you code

```
x <- seq(1,20,3)

x

[1]  1  4  7 10 13 16 19
```

x becomes the vector of all values down to the largest that fulfils the condition < to-by

Another type of sequence we may need to create is the repetition of elements. This is performed with the use of the rep() function

rep(x, ntimes)

```
y <- rep(1:3, 2)

y

[1] 1 2 3 1 2 3 #will produce a re-ordered matrix according to the above ordering
```

Other general functions

Obtaining a random set of values from a greater set is done with sample().

sample(x, size, replace=F/T)

sample() takes a subset of size=size from the vector x and returns it a smaller vector. If replace=T then the same element can be drawn more than once.

pretty(c(start,end), N)

returns a vector of equally spaced N values between start and end. Invaluable for simulating distributions, producing values for standard reference plots etc.

sort(x)

returns the vector x sorted in numerical order from the smallest to the largest element while

order(m[,c(i,j)])

orders a matrix m according first ot the values in the i-th and then in the j-th columns. Calling

m[order(m[,c(i,j)]),]

will produce a re-ordered matrix according to the above ordering

Input/Output: Reading data

R is able to handle great chunks of data in various levels of organization and the best way to feed them is by making R read them from a file stored in our computer. There are numerous ways to do so depending on the format, size and type of the data as well as on the downstream analyses we intend to conduct. In the following we will take a look at the most common ones. Simply invoke it with:

```
data<-read.table("myfile.txt")
```

Keep in mind that myfile.txt needs to be in the directory you are currently working in. In any other case you will need the full path of the file such as e.g.

```
data<-read.table("~/Documents/R/myfile.txt")
```

R will read through the file, skip any line starting with a comment hash "#" and will try to store the values read in the most convenient form, which is usually a data frame.

We can now check what the data frame holds by asking R to return the first rows using head() or the last ones using tail()

```
head(data)
```

```
tail(data)
```

Input/Output: Reading data

Reading through big files can take time even with R (or especially with R) as it tries to make some inference on the data "on the fly" as the file is being read. Both read.table and read.delim allow us to provide additional information before reading the file. In particular there are some important attributes/options we can activate that are related to:

- the column separator sep
- the header of the columns header
- the number of rows nrow

Lets try to let R know that we want to read a file using tab as the column separator, keeping the first line of the file as column header and reading 1000 rows

```
data<-read.delim("myfile.txt", header=T, sep="\t", nrow=1000)
```

This will keep the first line as column headers and will stop reading after the first 1000 lines (excluding the header). Each column will be read if it is tab-separated by the previous one. There are a number of ways to separate columns in tables, the most common ones being space, tab and commas. You may often (or not so often) see filenames ending with the *.tsv or *.csv extensions. These indicate tab-separated-values or comma-separated-values. R has a particular function to read the latter called read.csv()

```
data<-read.csv("myfile.csv")
```

will read and store data directly in columns as long as they are comma separated.

Input/Output: Writing data

We saw how we feed data into R, how we make sure non-sensical values are not included but now how about getting data out of R and into a file in our computer? R has specific functions for writing data to files, most of which are perfectly symmetrical to the reading ones. Thus if we want to write output to a file we can make use of the write function:

```
write(data, file="out.txt")
```

which can be made much more elaborate if additional options are fed in the command

```
write(data, file="out.txt", append=T, sep="\t", ncolumns=3)
```

this will not only write the "data" to a file names "out.txt" but it will further append the data at the end of this file if it already exists. Moreover, data will be written in 3 columns separated with tab.

Although the command above will work most of the times, it is probably safer to use write.table instead for increased control of the process.

```
write.table(data, file="out.txt", append=T, sep="\t", row.names=F,
col.names=T)
```

This is only to be used on a data frame or a matrix that already has the values spread out in rows and values (notice that there is no option for number of columns as there are predefined by the dataframe structure). row.names=F (meaning FALSE) tells R to skip enumerating the rows, since this usually adds and extra column to the output file (try this yourselves).

Writing to files can also be done in csv mode with (you might have guessed already) write.csv().

Basic Plotting

R has a huge toolbox of functions and packages for plotting simple data, time series, big data, complex data structures, maps etc.

Most importantly this toolbox is ever-expanding due to **R being open source**, which means that there is a constant flow of new functions for intricate plots and graphics in the repositories.

R's way of producing graphs differs from programs you may have used before (such as Excel or Graphpad) in the sense that **it is not so much based on a graphical user interphase as it is on the command line**. This may sound like a drawback in the beginning but once you get accustomed with the process of calling plot functions and customizing them on the command line you can store everything in custom scripts that will produce elegant, complicated graphs with the strike of a button.

Nevertheless, even the most elegant graphs are based on some very simple rules.

Barplots

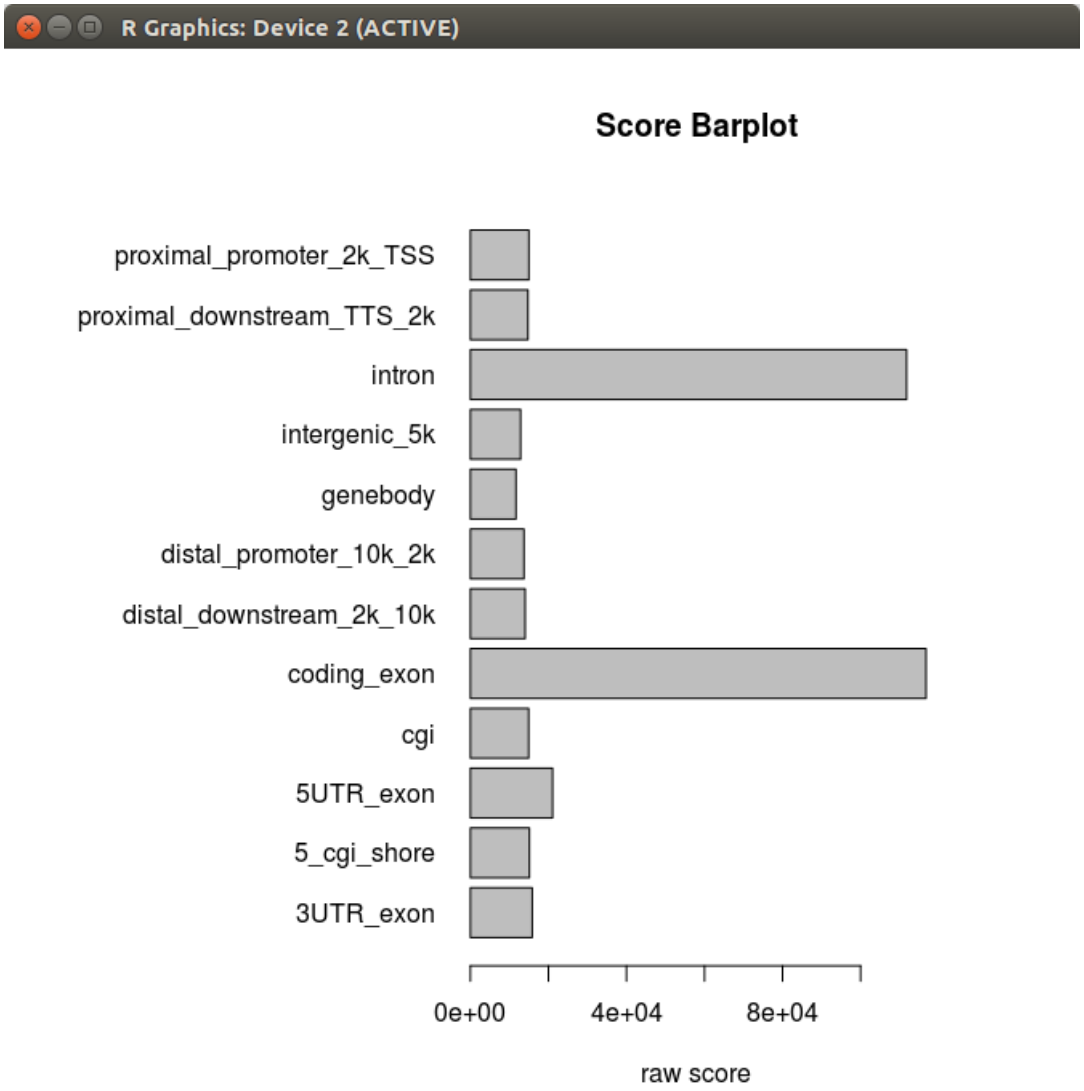
Plot one variable for various categories

```
data<-
read.delim("genome_par
tition.tsv", sep="\t",
header=T)

x<-data[,2]

par(mar=c(5,15,5,5))

barplot(data[,2],
names=data[,1],
horiz=T, las=1,
main="Score Barplot",
xlab="raw score")
```



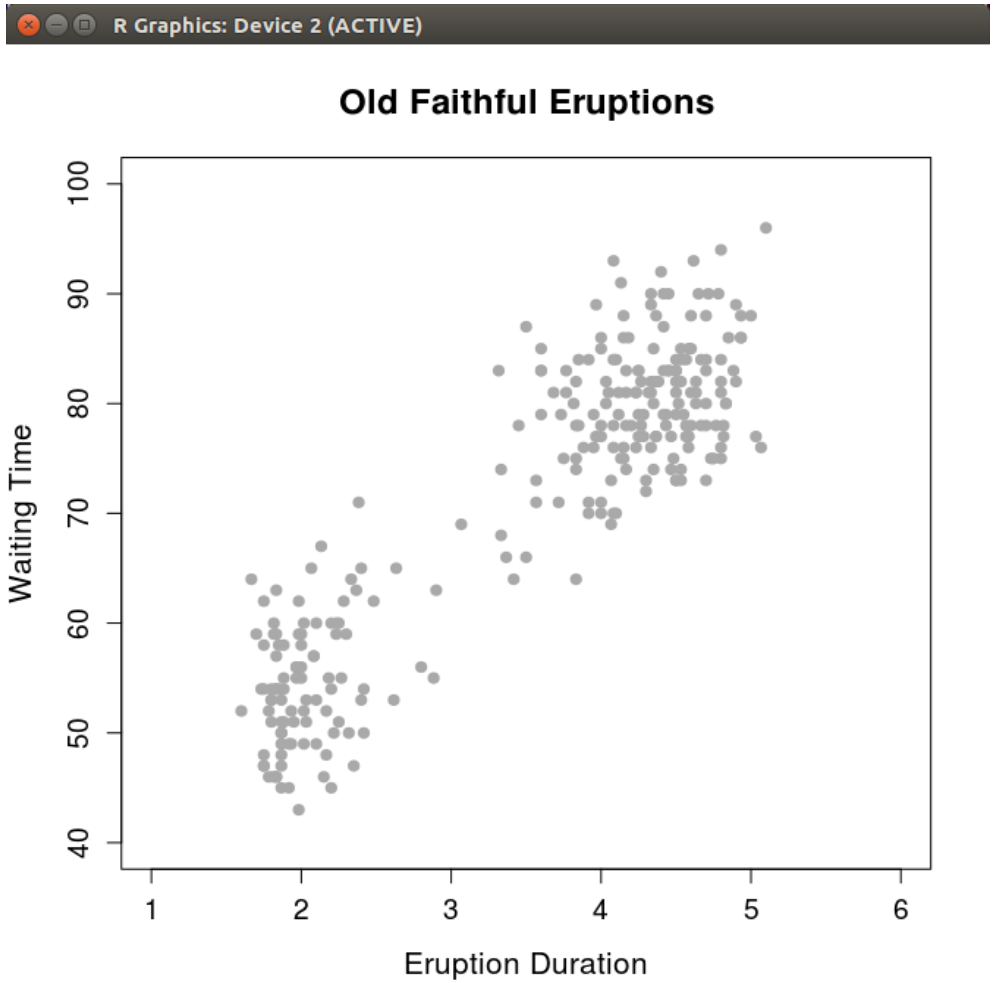
ScatterPlots

Plot two variables x,y against each other

```
data(faithful)

head(faithful)

plot(faithful$eruptions,
faithful$waiting,
col="dark grey", pch=19,
xlim=c(1,6),
ylim=c(40,100),
xlab="Eruption
Duration", ylab="Waiting
Time", main="Old
Faithful Eruptions",
cex.axis=1.2,
cex.lab=1.3,
cex.main=1.5)
```

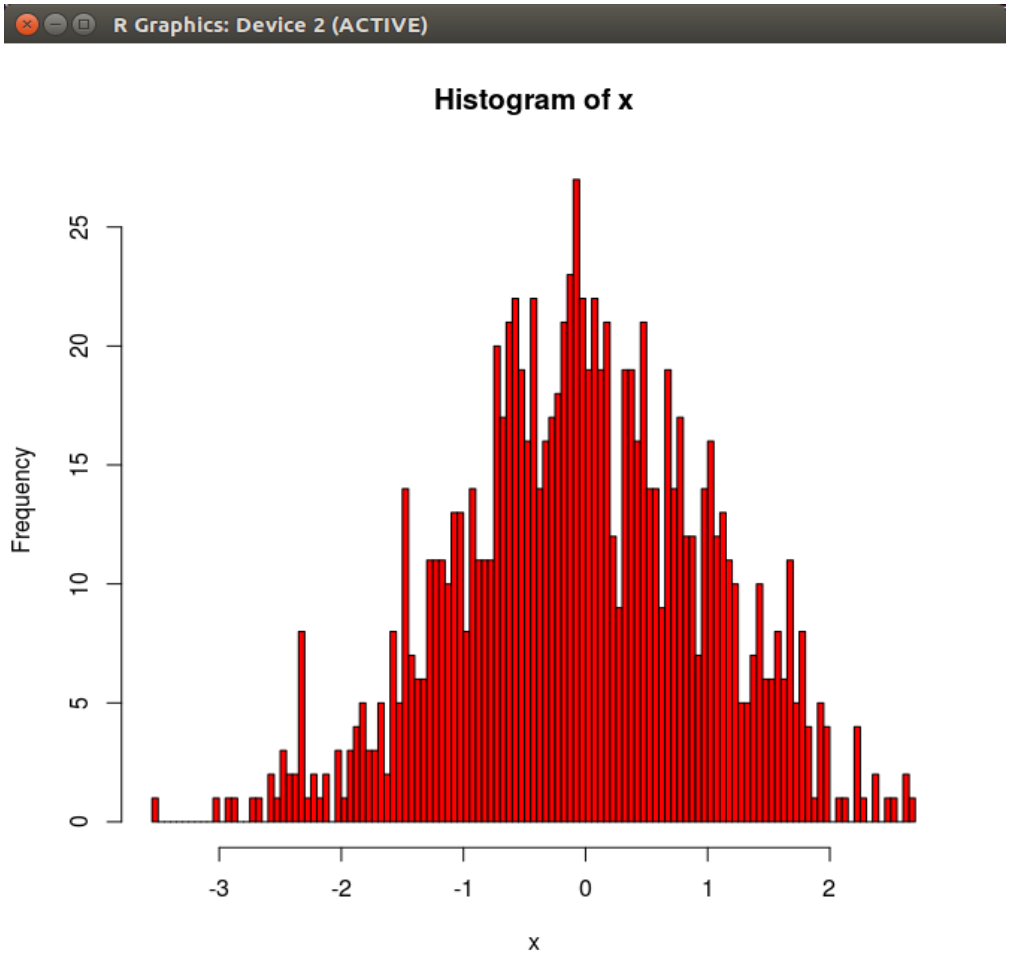


Histograms

Plot the histogram of a given numerical value

```
x<-rnorm(1000)

hist(x,
breaks=100,
freq=T,
col="red")
```



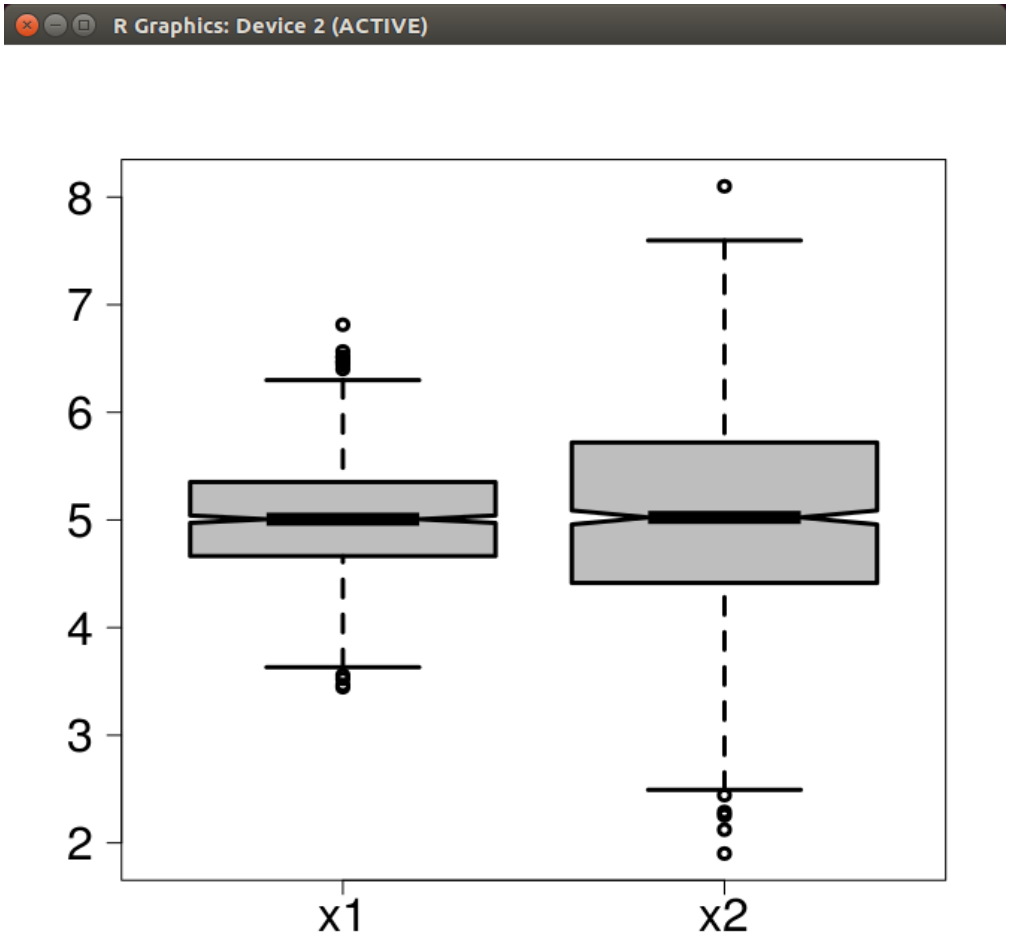
Boxplots

Compare two or more distributions with a "summarized" histogram in the form of a boxplot

```
x1<-rnorm(1000,
mean=5, sd=0.5)

x2<-rnorm(1000,
mean=5, sd=1)

boxplot(x1, x2,
col="grey", notch=T,
lwd=3,
names=c("x1","x2"),
cex.axis=2, las=1)
```



Print plots

R employs a simple routine to export plots in various image formats such as *jpeg,*png, *ps, *tiff, or *pdf.

The names of the functions are (again) self-explanatory. The grammar is the same in all of them and follows the simple structure

nameoffunction("filename"); plotting commands; dev.off()

An example printing of a plot to a pdf file may be:

```
pdf("boxplot.pdf")

boxplot(x1, x2, col="grey", notch=T, lwd=3,
names=c("x1","x2"), cex.axis=2, las=1)

dev.off()
```

Notice how it also works with the commands being issued in different lines.

Statistics in R

A great number of users, use R solely for statistical calculations and analyses. Although there is much more to R than statistics, the existence of an extensive (and ever-expanding) statistical toolbox remains one of its main strong points compared to other programing languages.

R has a plethora of both elementary and advanced statistical functions, including mode calculations, statistical tests, simulation of data etc. Below we start discussing some of the most basic functions that will help you go through some initial basic statistics with your data.

Descriptive Statistics

sum(x)

The sum() function returns the sum of a numerical vector

min(x), max(x), range(x)

Return the minimum and maximum values of numerical vectors. range(x) is the difference max-min

length(x)

The length() function applied to vectors (regardless of class) returns their size in number of elements. length() is thus the easier way of counting elements

mean(x, trim=0, na.rm=FALSE)

mean() is a function for calculating the mean (surprise!) of a vector of numerical values (keep in mind both "vector" and "numeric").

sd(x), var(x)

These return the standard deviation sd() and the variance var() of a numeric vector.

median(x)

The median of a numerical vector is the value that lies midway in a ranking of values from the lowest to the highest. Its difference from the mean is very often a good way to "guesstimate" on the existence of outliers and assymetric distributions

Descriptive Statistics

sum(x)

The sum() function returns the sum of a numerical vector

min(x), max(x), range(x)

Return the minimum and maximum values of numerical vectors. range(x) is the difference max-min

length(x)

The length() function applied to vectors (regardless of class) returns their size in number of elements. length() is thus the easier way of counting elements

mean(x, trim=0, na.rm=FALSE)

mean() is a function for calculating the mean (surprise!) of a vector of numerical values (keep in mind both "vector" and "numeric").

sd(x), var(x)

These return the standard deviation sd() and the variance var() of a numeric vector.

median(x)

The median of a numerical vector is the value that lies midway in a ranking of values from the lowest to the highest. Its difference from the mean is very often a good way to "guesstimate" on the existence of outliers and assymetric distributions

Correlation / Regression

We can calculate the correlation of two variables x, y with various measures with the cor() function.

cor(x,y, method = c("pearson", "kendall", "spearman"))

For which we can also calculate the significance of the obtained correlation value

cor.test(x,y, method=....)

Regression can be performed in various ways. The simplest one is linear regression which is performed with the lm (linear modeling) function. In the example that follows we test how the horsepower of a set of cars can be modeled on the weight of the car:

```
model<-lm(mtcars$hp~mtcars$wt)

model # will printCall:

lm(formula = mtcars$hp ~ mtcars$wt)

Coefficients:

(Intercept)    mtcars$wt

     -1.821       46.160
```

The model above prints two values that have to be explained as the a, b in a linear function of the general type:

f(x)=ax+b

Probability Functions and Simulation

The basic format for probablility functions in R is **[dpqr]-type of distribution**

d=density, p=probability function, q=quantile function and r=random generation

Out of the four, r is the one to be used more often providing us with a numerical vector of random values based on the type of distribution and the chosen parameters.

Some of the most common types of distributions are:

rnorm() for the normal distribution which can be called with

```
rnorm(n, m=0, sd=1)
```

which returns m values sampled from normal distribution with mean=0 and sd=1

rbinom() for the binomial distribution

which is called as

```
rbinom(n, N, prob=p)
```

runif() for the uniform distribution

which you can use with

```
runif(n, min=a, max=b)
```

to obtain n random elements sampled from a uniform distribution between a and b. runif() is very often used for the generation of random real numbers

A lot more is out there

Advanced statistics: p-value calculations, hypothesis testing etc

Advanced plotting: Elaborate representations, 3-D plots, maps etc

Programming: Writing your own code to perform analyses