MAST20005/MAST90058: Week 2 Lab & R Reference

Goals: (i) Getting started with R and RStudio; (ii) Basic exploratory data analyses; (iii) Basic graphics.

This is much longer than the usual weekly lab sheet because it includes a substantial guide to get you started. You are not expected to absorb it completely in a single lab session. We suggest you skim it upon first reading and ensure you set aside at least 15 minutes to work through the lab exercises at the end in your first lab class, referring back to previous sections when required.

1 Introduction

R is a popular computational tool for statistical applications. An increasing number of researchers in many disciplines, including social and biomedical sciences, choose R for their work. R is both a programming language and a general-purpose computing environment withing which many classical and modern statistical procedures have been implemented.

R is often the vehicle of choice for research in statistical methodology. Its open source nature allows statisticians to contribute new methods in the form of packages which can be downloaded and installed for free at any time. Besides the ordinary packages supplied in the 'base' version of R, hundreds of contributed packages are available through the CRAN website.

The main advantages of R are the fact that it is freely available and that help is readily available online. It is similar to some other high-level programming packages such as MATLAB (which is not free), but more user-friendly than low-level programming languages such as C++ or Fortran. You can use R as it is, but for educational purposes we prefer to use the RStudio interface (also freely available), which has a usefully organized window layout and several extra features which make working with R easier.

2 Getting started

2.1 Install R and RStudio

Both R and RStudio are installed on the machines in your computer labs. If you wish to install R on your own personal computer, go the R Project homepage and follow the instructions. Likewise, to install RStudio, go to the RStudio homepage and follow the instructions.

2.2 RStudio window layout

- Bottom left: console window (also called command window). Here you can type simple commands after the `>` prompt and R will then execute your command. This is the most important window because this is where R actually does things.
- Top left: editor window (also called script window). Collections of commands (scripts) can be edited and saved. If you do not see this window, you can open it with `File > New R script`. Just typing a command in the editor window is not enough, it has to get into the command window before R executes the command. If you want to run a line from the script window (or the whole script), you can click `Run` or press `CTRL+ENTER` to send it to the command window.
- Top right: workspace / history window. In the workspace window you can see which data and values R has in its memory. You can view and edit the values by clicking on them. The history window shows what has been typed before.

- Bottom right: files / plots / packages / help window. Here you can open files, view plots (also previous plots), install and load packages or use the help function.

2.3 Working directory

Your working directory is the folder on your computer in which you are currently working. When you ask R to open a certain file, it will look in the working directory for it, and when you tell R to save a data file or figure, it will save it in the working directory. Before you start working, please set your working directory to where all your data and script files are or should be stored.

In the command window: `setwd("directoryname")`. For example:

```
> setwd("M:/MyStuff/R/")
```

Alternatively, from the RStudio menus you can go to `Session / Set working directory` or `Tools / Global options / General / Default working directory`.

Make sure that the slashes are forward slashes and that you don't forget the apostrophes. R is case-sensitive, so make sure you write capitals where necessary.

2.4 Packages

R can carry out a large number of statistical analyses and procedures. This functionality is organized into a number of 'packages'. With the standard installation, most common packages are installed. To get a list of all installed packages, go to the packages window or type `library()`.

Alternatively, in RStudio you can use the console packages window (bottom right). If the box in front of the package name is ticked, the package is loaded (activated) and can be used. There are many more packages available on the R website. If you want to install and use a package, for example the package called `triangle`, you should click install packages in the packages window and type 'triangle' or type the following in the command window:

```
> install.packages("triangle")
```

To load the package: check the box in front of 'triangle' or type the following in the command window:

```
> library(triangle)
```

3 Examples of R commands

3.1 Using R as a calculator

We can warm up by using R as calculator:

```
62 + 5 - 7 * 9 + 15/3 - 2^2  # ^ means "to the power of"

## [1] 5
```

Note that commands after the symbol `#` are not executed. This can be used to include comments to lines of code.

The `[1]` is where the output begins, showing the result of your calculation. The number '1' here refers to the fact that it is first number in the output. If there are many numbers in the output and it is too long to fit on one line (e.g. the result is a long vector, see below), numbers in brackets at the start of each line will tell you which position in the output vector is shown at the start of that line.

As with most calculators, R evaluates operations in the following order:

1. Power
2. Multiplication & division
3. Summation & subtraction

For example, the following two commands give different results:

```
2 + 3 / 4^3

## [1] 2.046875

2 + (3 / 4)^3

## [1] 2.421875
```

There are many mathematical functions that are already in R, ready to use. For example the exponential function, `exp()`, the natural logarithm, `log()`, logarithm in base 2, `log2()`, the square root, `sqrt()` and trigonometric functions such as `sin()` and `cos()`.

```
log(25)        # Logarithm of 25 using natural base e

## [1] 3.218876

sqrt(4)        # Square root of 4

## [1] 2

pi * 4^2       # Area of a circle of radius 4

## [1] 50.26548

a <- pi * 4^2  # Create the variable "a"
a              # Print "a"

## [1] 50.26548
```

The assignment operator, `<-`, represents an arrow pointing at the object receiving the value of the expression. Instead of `<-`, some people prefer to use `=`, which is more standard in other programming languages. Either can be used in R.

To remove all variables from R's memory, type

```
rm(list = ls())
```

or click `clear all` in the workspace window. You can see that RStudio then empties the workspace window. If you only want to remove the variable `a`, you can type `rm(a)`.

3.2 Vectors and matrices

R's most basic way of storing data is as a vector (a 'row' of numbers). This is also known as a 1-dimensional array. It can also arrange data into multi-dimensional arrays. The special case of a 2-dimensional array is called a matrix.

To construct a vector named `x`, use the R command `c()` (it means "concatenate") as follows:

```
x <- c(4.1, -1.3, 2.5, -0.6, -21.7)
x

## [1]   4.1  -1.3   2.5  -0.6 -21.7
```

The following creates a new vector `z` by combining two vectors:

```
x <- c(1, 3, 9, 10)
x

## [1]  1  3  9 10

y <- c(30, 35, 4)
y

## [1] 30 35  4

z <- c(x, y)
z

## [1]  1  3  9 10 30 35  4
```

R enables us to generate commonly used sequences of numbers. For example `1:30` is the vector `c(1, 2, ..., 29, 30)`.

```
n <- 5
1:n - 1

## [1] 0 1 2 3 4

1:(n - 1)

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

Comparing the two sequences tells us that the priority of the colon operator '`:`' is higher than the negation operator '`-`' (i.e. the colon gets executed first). In general it has higher priority than any other arithmetic operator ('`+`', '`-`', '`*`', '`/`').

Elements in vectors can be addressed by standard `[i]` indexing. For example:

```
z[2]

## [1] 3

z[1:4]

## [1]  1  3  9 10

z[c(1, 2, 6)]

## [1]  1  3 35

z[2] <- 2015
```

In the last line, one of the elements is replaced with a new number.

The function `seq()` is another useful tool for generating sequences. The first two arguments, if given, specify the beginning and end of the sequence.

```
seq(1, 2, by = 0.1)

##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0

seq(1, 2, length.out = 20)

##  [1] 1.000000 1.052632 1.105263 1.157895 1.210526 1.263158 1.315789
##  [8] 1.368421 1.421053 1.473684 1.526316 1.578947 1.631579 1.684211
## [15] 1.736842 1.789474 1.842105 1.894737 1.947368 2.000000
```

A matrix is can be created from scratch using the function `matrix()`. For example, to define the matrix

$$A = \begin{pmatrix} 2 & 4 & -1 \\ -1 & 2 & 3 \end{pmatrix}$$

you can use the function `matrix()` on a vector where the elements of A are listed column by column:

```
A <- matrix(c(2, -1, 4, 2, -1, 3), nrow = 2) # create a matrix
A

##      [,1] [,2] [,3]
## [1,]    2    4   -1
## [2,]   -1    2    3
```

Matrix operations are similar to vector operations:

```
A[1, 2]   # extract the element in first 1st row and 2nd column

## [1] 4

A[1, ]    # extract the first row

## [1]  2  4 -1

mean(A)   # sample average for all the elements

## [1] 1.5
```

Elements of a matrix can be addressed in the usual way: `[row, column]`. When you want to select a whole row, you leave the spot for the column number empty (the other way around for columns of course). The last line shows that many functions also work with matrices as arguments.

3.3 Characters

Character vectors are used frequently in R, for example as plot labels. When needed they can be entered by quotes characters. For example:

```
u1 <- c("male", "female")
u2 <- c("apple", "pear", "kiwi", "orange")
u1

## [1] "male"   "female"

u2

## [1] "apple"  "pear"   "kiwi"   "orange"
```

A useful function to combine numeric and character vectors is `paste()`, which takes an arbitrary number of arguments and concatenates them one by one into character strings. The arguments are, by default, separated in the result by a single blank character, but this can be changed using the named parameter `sep`. For example:

```
labels <- paste(c("X", "Y"), 1:10, sep = "")
labels

##  [1] "X1"  "Y2"  "X3"  "Y4"  "X5"  "Y6"  "X7"  "Y8"  "X9"  "Y10"
```

3.4 Vectorisation

When you ask R to do a calculation with a vector, it will usually repeat the same operation on each element of the vector. For example, the following does elementwise addition:

```
c(1, 2) + c(2, 5)

## [1] 3 7
```

Such operations are called *vectorised* operations. Many functions and operations in R are vectorised, which usually leads to neater and more concise code.

When the vectors are all of the same length, then the operation is well-defined: R simply applies the operation to each element in turn. If you give R vectors of different lengths, then it will 'loop back' on the shorter vectors and start repeating elements from the start. This is called *recycling* the vector. For example:

```
c(1, 2, 3) + c(2, 5)

## Warning in c(1, 2, 3) + c(2, 5): longer object length is not a multiple of shorter
object length

## [1] 3 7 5
```

Often this is not what you want, so R will sometimes print a warning message as it did above. However, one case where it is very handy is when you wish to recycle a single value:

```
1:5 + 3

## [1] 4 5 6 7 8
```

4 Functions

You can write functions to automate calculations or operations. Some functions are standard in R or in one of the packages. Later you will learn how to program your own functions. Important basic functions to know are:

- `max` and `min`: select the largest and smallest elements of a vector
- `range` is a function whose value is a vector of length two: `c(min(x), max(x))`
- `sort` returns a vector of the same size as the original vector with the elements arranged in increasing order
- `length` is the number of elements in a vector
- `sum` gives the total of the elements in vector
- `prod` gives the product of the elements in vector
- `dim` gives the dimensions of multi-dimensional array
- `mean` gives the sample mean of the vector, which is the same as `sum(x) / length(x)`

For example, to compute a sample mean:

```
mean(z)

## [1] 300.5714
```

Within the brackets you specify the arguments. Arguments give extra information to the function. In this case, the argument says which vector of numbers to compute the mean for.

The function `rnorm` is a standard R function which creates random samples from a normal distribution:

```
z <- rnorm(10)  # generate a vector with 10 observations from N(0,1)
z

##  [1]  0.60004985  0.48977383  0.38102583  1.25330446 -0.57493310
##  [6] -1.83025957  1.65287576 -2.19681254 -0.51336285 -0.03412181
```

Here `rnorm` is the function and the 10 is an argument specifying how many random numbers you want, in this case 10 numbers (typing `n = 10` instead of just `10` would also work). Note that entering the same commands again produces 10 new random numbers. Instead of typing the same text again, you can also press the upward arrow key '↑' to access previous commands.

If you want 10 random numbers out of normal distribution $N(\mu = 10, \sigma^2 = 2^2)$ you can type:

```
x <- rnorm(10, mean = 10, sd = 2)
x

##  [1]  6.320776  7.321068  9.567314 10.022107 11.834722 11.038895  7.461893
##  [8]  8.106456  7.729224  6.419676
```

Thus, we see that the same function (`rnorm`) may have different interfaces and that R uses *named* arguments (in this case `mean` and `sd`). By the way, the spaces around the '`,`' and '`=`' symbols do not matter. Comparing this example to the previous one also shows that for the function `rnorm` only the first argument (the number `10`) is compulsory, and that R gives default values to the others (these are known as *optional* arguments). RStudio has a nice feature: when you type `rnorm(` in the command window and press TAB, RStudio will show the possible options.

4.1 Getting help

There is a vast amount of (free) documentation and help available. Some help is automatically installed. Typing in the console window the command:

```
> help(boxplot)
```

gives the documentation on the `boxplot` function. It gives a description of the function, all of the possible arguments, the values that are used as default for each optional argument, and various other information. Typing

```
> example(boxplot)
```

gives some examples of how the function can be used.

5 Scripts

You can store your commands in files called scripts. These have the extension `.R`, e.g. `myscript.R`. You can open an editor window to edit these files by clicking `File` and `New` or `Open file...`. You can run (send to the console window) part of the code by selecting lines and pressing `CTRL+ENTER` or click `Run` in the editor window. If you do not select anything, R will run the line your cursor is on. You can always run the whole script with the console command:

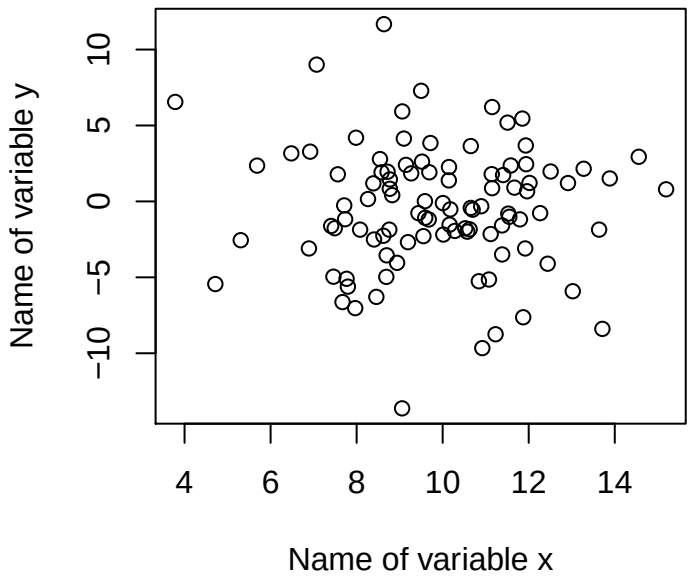
```
> source(myscript.R)
```

or press `CTRL+SHIFT+S`.

6 Graphics

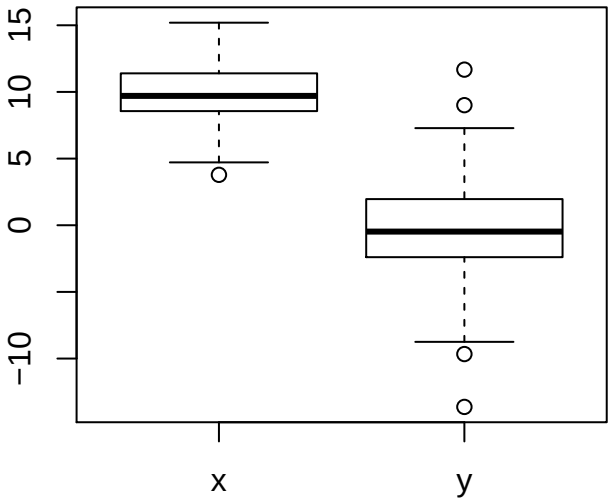
Visualising data is an important component of statistical modelling and data analysis. R can be used to make a large variety of different plots. Here is a simple example:

```
x <- rnorm(100, mean = 10, sd = 2)  # sample of size 100 from N(10, 4)
y <- rnorm(100, mean =  0, sd = 4)  # sample of size 100 from N(0, 16)
plot(x, y, xlab = "Name of variable x", ylab = "Name of variable y")
```



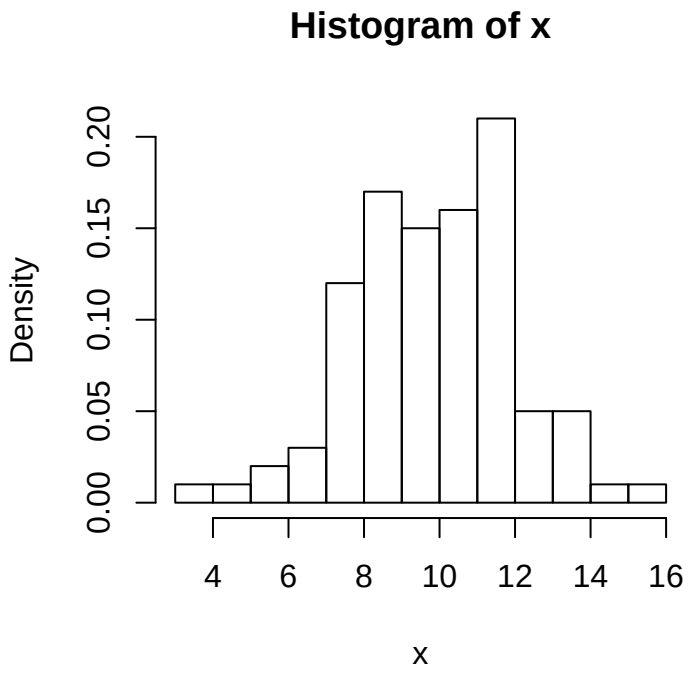
In the first two lines, random numbers are assigned to the variables `x` and `y`, which become vectors. In the third line, pairs of values are plotted in the plots window. The following creates a paired boxplot:

```
boxplot(x, y, names = c("x", "y"))  # creates paired boxplots
```



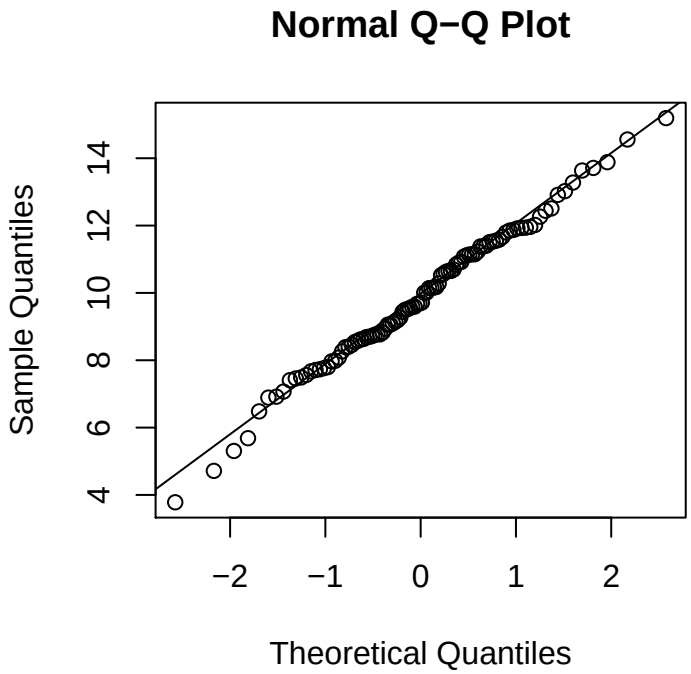
Other simple examples are the histogram and normal QQ plots, generated by the simple command:

```
hist(x, freq = FALSE, nclass = 10)
```



The second argument specifies that a density histogram is desired (`freq = TRUE` shows relative frequencies instead), while the third argument specifies the number of bins.

```
qqnorm(x)
qqline(x)
```



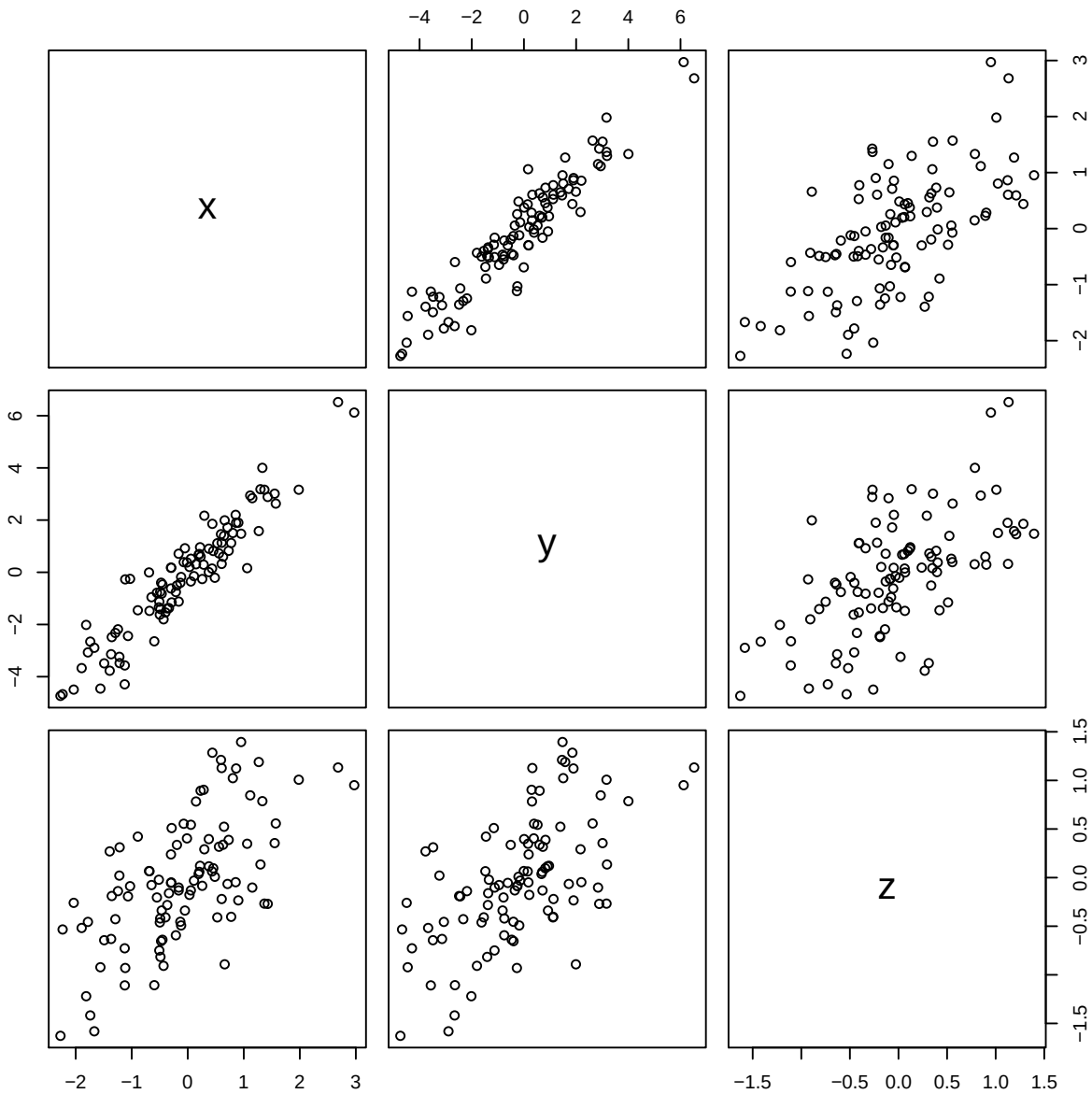
7 Other data structures

Datasets are often organised into data frames, which is like a matrix with names above the columns. This is nice, because you can call and use one of the columns without knowing in which position it is.

```
x <- rnorm(100)                    # generate 3 vectors of length 100
y <-   2 * x + rnorm(100, 0, 0.8)
z <- 0.5 * x + rnorm(100, 0, 0.5)
t <- data.frame(x, y, z)           # create a data frame
summary(t$x)                       # summary statistics for x

##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.27169 -0.65598 -0.09537 -0.07988  0.60522  2.97144

plot(t)                            # scatter plot matrix
```



Another basic structure in R is a *list*. The main advantage of lists is that the objects forming the list do not have to be of the same length. (Data frames are actually a special type of list.)

```
L <- list(one = 1, two = c(1, 2), five = seq(0, 1, length.out = 5))
L

## $one
## [1] 1
##
## $two
## [1] 1 2
##
## $five
## [1] 0.00 0.25 0.50 0.75 1.00

L$five + 10

## [1] 10.00 10.25 10.50 10.75 11.00
```

8 Reading and writing data

A simple way to load a data set from a text file or a URL in RStudio is `Tools > import dataset`. There are many ways to write data from within the R environment to files, and also to read data from files. The following lines illustrate some basic steps using console commands:

```
# Construct and store a simple data frame.
t <- data.frame(x = c(1, 2, 3), y = c(30, 20, 10))
t

##   x  y
## 1 1 30
## 2 2 20
## 3 3 10

write.table(t, file = "mydata.txt", row.names = FALSE) # save file
t2 <- read.table(file = "mydata.txt", header = TRUE)   # load file
t2

##   x  y
## 1 1 30
## 2 2 20
## 3 3 10
```

The argument `row.names = FALSE` prevents that row names are written to the fille. Nothing is specified about `col.names`, so the default option `col.names = TRUE` is chosen and column names are written to the file. When reading a file note that the column names are also read if `header = TRUE`. The data frame also appears in the workspace window.

R has several additional packages for reading external data format. For example, the function `read.csv()` reads CSV formatted text files. Another example is the package `foreign` which contains functions such as `read.dta()` & `write.dta()`, and `read.spss()` & `write.spss()`, for dealing with Stata and SPSS formats respectively.

9 Missing data

When you work with real data, you will encounter missing values. When a data point is not available, you write `NA` instead of a number.

```
x <- c(rnorm(10), NA, rnorm(2))
```

Computing statistics of incomplete data sets is usually an undefined operation, and the output itself will also be `NA`. For example, R will say that it doesn't know what the smallest value of `x` is:

```
min(x)

## [1] NA
```

If you do not care about missing data and want to compute the statistics with the values that are present, use the additional argument `na.rm = TRUE`:

```
min(x, na.rm = TRUE)

## [1] -2.191828

mean(x, na.rm = TRUE)

## [1] -0.1324518
```

10 Conditional execution and loops

In R, commands can be grouped together by braces, { `expression1; expression2; ...` }, and the result of the group expression is the result of the last expression being evaluated. We can use single expressions or groups of expression to construct conditional statements with the following syntax:

```
> if (expression1) expression2 else expression3
```

where `expression1` is a condition resulting in a logical value `TRUE` or `FALSE`. If `expression1` is true, then `expression2` is evaluated. If `expression1` is false, then `expression3` is evaluated instead.

Let us consider a sample of $n = 10$ observations from a normal $\mathrm{N}(0, 1)$,

```
x <- rnorm(10)
```

The following code checks whether the mean is greater than the median and prints a string with the outcome.

```r
if (mean(x) > median(x)) {
    "The mean is greater than the median"
} else {
    "The mean is smaller than the median"
}

## [1] "The mean is greater than the median"
```

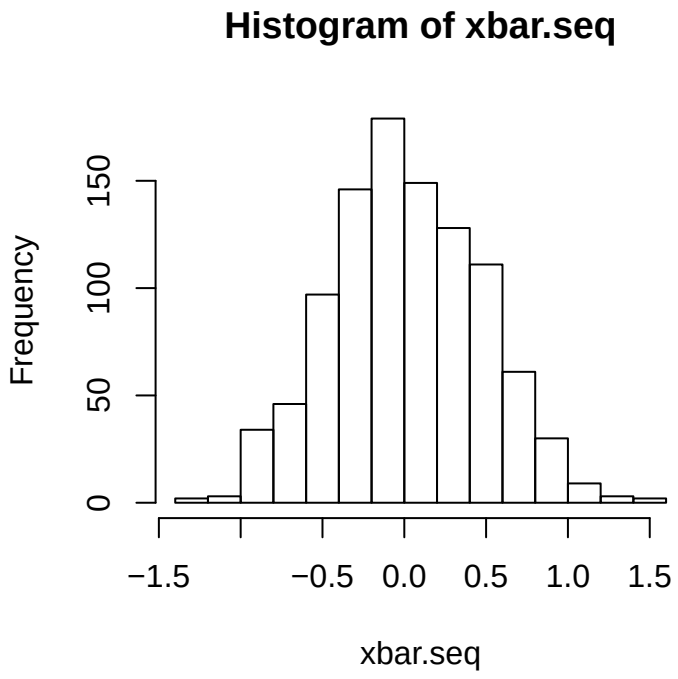
There is also a vectorised version of the if-else statement, in the form a function called `ifelse`. The syntax for it is `ifelse(condition, a, b)`.

Suppose we want to carry out a task a fixed number of times, say B. To this end, it is convenient to construct for-loops. In a for-loop, you specify a list of possible values and also the name of a variable, which will take each of those values in turn and perform a calculation.

```
> for (name in expression1) expression2
```

where `name` is the loop variable and `expression1` is a vector expression (in many cases a sequence like `1:100`). From example, consider the drawing $B = 1000$ samples of size $n = 5$ from a $\mathrm{N}(0, 1)$. For each sample, we compute $\hat{x}$, and store the results in a vector of length B:

```r
B <- 1000                          # number of runs
n <- 5                             # sample size
xbar.seq <- 1:B                    # a vector of size to be filled with means
for (i in 1:B) {
    sample <- rnorm(5)
    xbar.seq[i] <- mean(sample)
}
hist(xbar.seq)                    # plot the results
```



11 Write your own functions

The R language allows the user to create new functions. These are true R functions that are stored in a special internal form and may be used in further expressions and so on. In the process, the language gains enormously in power, convenience and elegance, and learning to write useful functions is one of the main ways to make your use of R comfortable and productive.

A function is defined by an assignment of the form:

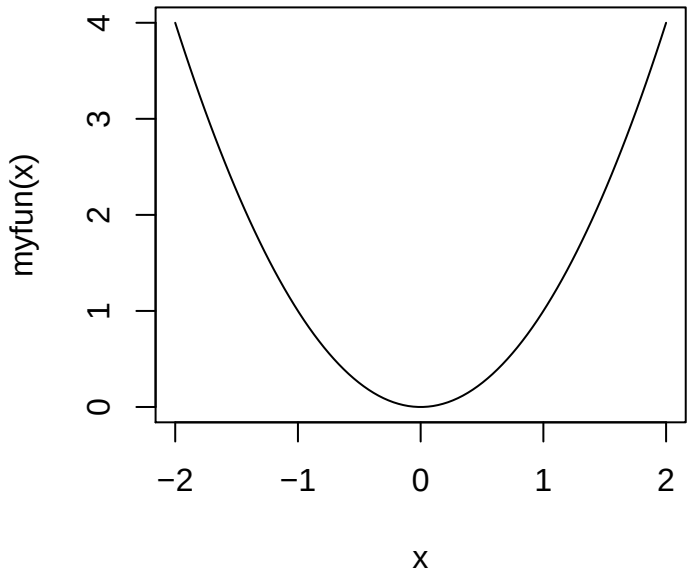
```
> namefunction <- function(argument_1, argument_2, ...) { expression }
```

For example

```
myfun <- function(x) {          # Specifies function name and argument
    y <- x^2                    # Specifies what the function should do
    return(y)                   # Returned value
}
myfun(1.5)                      # Computes 1.5^2

## [1] 2.25

x <- seq(-2, 2, length.out = 100) # Plots the new function
plot(x, myfun(x), type = "l" )
```



Next write a function to compute the median. A simple algorithm is to sort the data and take the middle element:

```
mymedian <- function(x) {
    n <- length(x)
    m <- sort(x)[(n + 1) / 2]
    return(m)
}
```

Note that this function operates without an error message for n even or odd, but does not return the correct result for even n. For even sample sizes, we should take the average of the two middle values:

```
mymedian <- function(x) {
    n <- length(x)
    if (n %% 2 == 1) { # odd
        med <- sort(x)[(n + 1) / 2]
    } else {   # even
        middletwo <- sort(x)[(n / 2) + 0:1]
        med <- mean(middletwo)
    }
    return(med)
}
x <- rnorm(10)
mymedian(x)

## [1] -0.2595761

median(x)

## [1] -0.2595761
```

Our new function seems to operate correctly and gives a results matching the default `median` function.

Exercises

1. Calculate $\sum_{i=1}^{100} \ln(i)$.
2. Let $X \sim \mathrm{N}(1, 2)$. What is $\mathbb{E}(X^2)$? Approximate this by simulating a large number of normal random variables and doing an appropriate calculation.
3. Use the help system to find out what the `qnorm` function does. Explain the result of running `qnorm(0.1)`. What about `qnorm(0.1, lower.tail = FALSE)`? For what value of `a` will the command `qnorm(a)` return the same value as the previous one?
4. Write a function `exp1pdf` that calculates the pdf of an exponential distribution with mean 1. Compare the output of your function against the in-built function that does the same calculation (`dexp`). Remember to check it works for all inputs: for example, what is the correct value of `exp1pdf(-1)`?
5. Do question 6(c) from the tutorial problems.
6. Do question 7(b) from the tutorial problems.
7. Do question 2(c) from the tutorial problems by simulation (rather than using the Central Limit Theorem).