1 Introduction

This document has no pretension of being comprehensive. The aim is just to help you to start with R. You will learn more during the other courses, by using the embedded help system and by reading the documents presented in the section\footnote{under windows all you need to do is to double-click on the executable you downloaded. The install program will guide you through}. The section\footnote{will explain some of the terms used. You will find those terms in bold throughout this document. For the sake of clarity some aspects are probably exaggerately simplified. It is hoped this causes no harm.} will explain some of the terms used. You will find those terms in bold throughout this document. For the sake of clarity some aspects are probably exaggerately simplified. It is hoped this causes no harm.

1.1 History and availability

S was developed at Bell laboratories in 70s. It became soon S-plus. R was started as a free implementation of the S/S-plus language. The two languages remain mainly identical.

The source code for R is freely available can be compiled for a very wide range of platforms. Executable binaries are also available for platforms like Windows (98, 2000, XP, NT), MacOS (version IX and X), and several linux distributions. I have a computer at home, you should be able to run R easily.

1.2 Why should we use R ?

Software based on spreadsheets (Microsoft’s Excel is a popular one) are not the only way handle data. The R software will let you do all what you were doing, but will also let you go (far) beyond. A large number of packages written by renowned statisticians are available, and you will be able to write your own code.

1.3 Starting (and exiting) R

The installation procedure will not be detailed\footnote{under windows all you need to do is to double-click on the executable you downloaded. The install program will guide you through}.

With Unix/Linux, just write R. Your terminal is now a R console. With Windows, double-click on the R icon. An R process is started. A R console window is opened. The console shows a > waiting for your input.

R has a powerful system to query help. By entering \texttt{help.start()} a browser will be started. You can navigate with the mouse through the help. Once this has been done, the results of help calls will be returned to the browser. The command \texttt{help} queries help. (\texttt{help()} is equivalent to \texttt{help(help)}).
To exit, write q(). You are prompted to known if you want to save your working session. Answering y or n will lead you out.

1.4 Interactive browsing of examples

This document can be used interactively within R. To do so, you will need:

1. the packages DynDoc and tkWidgets installed
3. to enter the R code at http://www.cbs.dtu.dk/staff/laurent/teaching/crashR_start.R or enter blindly the following command in R

source(url("http://www.cbs.dtu.dk/laurent/teaching/crashR_start.R"))

2 Basics about the language

2.1 Like a pocket calculator (just bigger)

R is designed to do statistics, hence to handle numbers. You can enter the following line:

> 1 + 2

[1] 3

The results is 3, as one might have expected.

"+" is called an arithmetic binary operator. It is arithmetic because it does some maths. It is binary because it takes two arguments (1 and 2). It is an operator because it does something.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiply (product)</td>
</tr>
<tr>
<td>/</td>
<td>division. Divide the first argument by the second argument</td>
</tr>
<tr>
<td>^</td>
<td>power. Multiply the first argument the number of times given in the second argument. ( x^y ) writes ( x \ ^ y ) in R.</td>
</tr>
<tr>
<td>%%</td>
<td>modulo (remainder of the integer division). Example: 10 ( %% ) 3 returns 1.</td>
</tr>
<tr>
<td>%%</td>
<td>integer division. Example: 10 ( %%% ) 3 returns 3.</td>
</tr>
</tbody>
</table>

Operators have a precedence (i.e. a relative priority). The parenthesis can be used to indicate in which order the computation should be performed.

> 2 + 3 * 4

[1] 14

> 2 + (3 * 4)
Many mathematical functions are also available. Example: To name few of them:

- **Trigonometric functions**
  - `cos`, `acos` cosine, arc-cosine
  - `sin`, `asin` sinus, arc-sine
  - `tan`, `atan` tangent, arc-tangent

- **Logarithm and exponentials**
  - `log` logarithm.
  - `log2` base-2 logarithm (\( \log_2 x \) is done in \( R: \log2(x) \)).
  - `exp` exponential (\( \exp x \) is done in \( R: \exp(x) \)).

- **miscellaneous**
  - `sqrt` square-root (\( \sqrt{x} \) is done in \( R: \sqrt{x} \)).

```r
> cos(pi/3)
[1] 0.5
```

### 2.2 variables (objects)

An R environment can be thought of as a working space. Naturally you can store things (eventually to re-use them later) in it. The things or objects you store are like boxes in a storage room. To find something, or actually to ask \( R \) to give you something, it is convenient to have a name for it. The "<-" operator performs what is called **assignment**. The first **argument** is a name for the **object** given in the second **argument**. If one does

```r
> x <- 1
```

the numerical value 1 is stored under the name \( x \). To query this object, just call it by its name:

```r
> x
[1] 1
```

In \( R \), a copy of the object is made during an assignment:

```r
> print(x)
[1] 1
```

```r
> y <- x
> print(x)
[1] 1
```
Variables are very convenient to store intermediate results. Example:

```r
> alpha <- 0.34
> a <- cos(alpha)
> b <- sin(alpha)
> (a^2) + (b^2)
[1] 1
```

The example above probably reminded you of something:

\[ \forall \alpha, \cos^2 \alpha + \sin^2 \alpha = 1 \]

The function `objects()` returns the names of the objects in an `environment` (or working space).

```r
> objects()
[1] "a"  "alpha" "b"  "x"  "y"
```

The function `rm(object)` removes the object (i.e. deletes it).

```r
> foo <- 33
> objects()
[1] "a"  "alpha" "b"  "foo" "x"  "y"
> rm(foo)
> objects()
[1] "a"  "alpha" "b"  "x"  "y"
```

### 2.3 mode (or type)

The objects we have seems were all numbers, but we can manipulate more than numbers.

```r
> name <- "George"
> age <- 43
> sex <- factor("MALE", levels = c("MALE", "FEMALE"))
> married <- TRUE
```
The type of the object is also called the mode in R. Knowing what an object is important for the system to know how to handle it. Example:

\texttt{name + age}

returns:

\texttt{Error in name + age : non-numeric argument to binary operator}

\textit{R} did not know how to perform "+" using a \texttt{character} and a \texttt{numeric}.

The different modes are:

\begin{itemize}
  \item The mode \texttt{character} is for strings.
  \item The mode \texttt{numeric} deals with \textit{real} numbers.
  \item The mode \texttt{integer} concerns integers.
  \item The mode \texttt{logical} would be called \textit{boolean} in other languages. It is linked to something called \textit{boolean logic}. Boolean operators are:
    \begin{verbatim}
    x && y  x AND y
    x || y  x OR y
    ! x     NOT x
    xor(x, y)  x XOR y
    \end{verbatim}
    They can be combined to verify

    \begin{verbatim}
    > has.plane.ticket <- TRUE
    > has.VISA <- TRUE
    > can.travel <- has.plane.ticket & has.VISA
    > print(can.travel)
    [1] TRUE
    > cannot.travel <- (!has.plane.ticket) | (!has.VISA)
    \end{verbatim}
  \end{itemize}

\item The mode \texttt{factor} is for categories. A factor can have different \textit{levels}.

\item The mode \texttt{list}. More will taught in Section 2.5.

\subsection*{2.4 vectors, matrices, arrays}

Some have probably already noticed a [1] in the front of the \textit{R} output. This means that what is returned on this line starts with the element number 1. \textit{R} is oriented to handle \texttt{vectors}. A \texttt{vector} can be thought of as sequence of elements of the same \texttt{mode}. What we manipulated so far were just vectors of length 1. Let’s see with an example:

\begin{verbatim}
> x <- c(1, 2, 3)
> print(x)
[1] 1 2 3
> y <- c("a", "b", "c")
> print(y)
\end{verbatim}
The function `c()` concatenates all its arguments into one vector. Other useful functions are `seq` and `rep`:

```r
> seq(1, 3)
[1] 1 2 3
> 1:3
[1] 1 2 3
> seq(1, 3, by = 0.5)
[1] 1.0 1.5 2.0 2.5 3.0
> rep(1, 3)
[1] 1 1 1
> rep(1:3, 2)
[1] 1 2 3 1 2 3
```

Many functions in R are designed to operate on vectors.

```r
> x <- c(-1, 2, 3)
> min(x)
[1] -1
> max(x)
[1] 3
> mean(x)
[1] 1.333333
> x + x
[1] -2 4 6
```

You can try with other functions defined in the section 2.1.

One important thing in R is what is called the *recycling rule*. Vectors that are shorter than they should see their elements *recycled* when looping through.

```r
> x <- c(-1, 2, 3, 3)
> x + 1
[1] 0 3 4 4
> x + c(1, 2)
[1] 0 4 4 5
```
> x = mean(x)

[1] -2.75 0.25 1.25 1.25

Many other functions than c() prove useful when dealing with vector. Some are:

- **rep(x, y)** repeat x y times.
- **seq(x, y)** generate a sequence of integers from x to y. A syntactic sugar for rep() is :: 1:10 is equivalent to rep(1,10).
- **rev(x)** returns x in the reverse order
- **sort(x)** returns x sorted in ascending order.

Matrices are very convenient objects to make computation. They are very similar to vectors.

> m <- matrix(c(1, 2, 3), nrow = 3, ncol = 3)
> m

```
[,1] [,2] [,3]
[1,] 1 1 1
[2,] 2 2 2
[3,] 3 3 3
```

> m + m

```
[,1] [,2] [,3]
[1,] 2 2 2
[2,] 4 4 4
[3,] 6 6 6
```

Some of the useful matrix facilites are:

<table>
<thead>
<tr>
<th>matrix multiplication</th>
<th>transpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>m %*% m</td>
<td>t(m)</td>
</tr>
<tr>
<td>diagonal elements</td>
<td>diag(m)</td>
</tr>
<tr>
<td>eigen values</td>
<td>eigen(m)</td>
</tr>
</tbody>
</table>

### 2.5 list and data frames

A list is also a vector of mode list, but the content of a list can be anything. This allows to bundle together heterogeneous things.

> list(1:10, c("a", "b", "c", "d"))

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

[[2]]
[1] "a" "b" "c" "d"
```

Elements in a list can be named:

> list(a = 1:10, b = c("a", "b", "c", "d"))
A **data.frame** is a list with a convenient particularity: each of its *components* are vectors of the same length. You can picture it as a table having each column filled with elements of the same *mode*. This is particularly convenient to store most of the datasets:

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>sex</th>
<th>married</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>43</td>
<td>MALE</td>
<td>TRUE</td>
</tr>
<tr>
<td>Anna</td>
<td>14</td>
<td>FEMALE</td>
<td>FALSE</td>
</tr>
<tr>
<td>Sarah</td>
<td>58</td>
<td>FEMALE</td>
<td>FALSE</td>
</tr>
<tr>
<td>Tom</td>
<td>35</td>
<td>MALE</td>
<td>FALSE</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Objects **data.frame** can be *attached* and *detached* (functions `attach()` and `detach()` respectively). Their content becomes then directly accessible.

### 2.6 subsetting, indexing

The objects we have presented have several elements. The access to one or more of these elements is called *subseting*. It is a very efficient technique in *R*. We will show how to do it with vectors, matrices, lists and data frames.

```r
> x <- c(2, 0, 1, 3, 4)
> x[1]
[1] 2
> xIndex <- c(1, 3)
> x[xIndex]
[1] 2 1
```

The subsetting operator is "[". A vector is used to tell which elements to select. A **logical** vector can also be used to select elements.

```r
> x[c(TRUE, FALSE, TRUE, FALSE, FALSE)]
[1] 2 1
> isSmall <- (x < 3)
> x[isSmall]
[1] 2 0 1
```

```r
> indexSmall <- which(isSmall)
> x[indexSmall]
[1] 2 0 1
```
For matrices, we have to specify two vectors of indices

```r
> m <- matrix(1:9, nrow = 3, ncol = 3)
> m

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

> m[1, 1]
[1] 1

> m[1, ]
[1] 1 4 7

> m[c(1, 2), c(1, 2)]

    [,1] [,2]
[1,]  1  4
[2,]  2  5

> indexDiag <- matrix(rep(1:3, 2), nr = 3, nc = 2)
> m[indexDiag]

[1] 1 5 9
```

We saw in Section ?? that a list can have named elements. The subsetting can be done by the name, using the operator [ with name or a number. Access to elements is done with the operator [[ or the operator "$"

```r
> l <- list(letter = c("a", "b"), number = c(1, 2, 3, 4))
> l

$letter
[1] "a"  "b"

$number
[1] 1 2 3 4

> l[\"letter\"]

$letter
[1] "a"  "b"

> l[1]

$letter
[1] "a"  "b"

> l$letter
```

9
For data frames:

```r
> df <- data.frame(name = c("George", "Anna"), age = c(43, 14))
> df

  name age
1 George 43
2 Anna  14

> df[1, 1]
[1] George

Levels: Anna George
```

By default, vectors of character are converted to factor in data.frame.

Subsetting by name can be done with any kind of vector, given that its elements were previously named. The operator [[ introduced for the list subsets and does not keep the name.

```r
> x <- c(2, 0, 1, 3, 4)
> names(x) <- c("a", "b", "c", "d", "e")
> x[3]

1

> x["c"]

1

> x[[3]]

[1] 1
```

### 2.7 Exercises

- compute with *R*:
  \[
  \frac{e^{\sin(\pi/16)}}{1 - e^{\sin(\pi/16)}}
  \]

- generate a vector *x* of 30 random numbers (normal distribution) using the function `rnorm`.

- create a vector *y* that only contains the positive elements of *x*.

- create a vector *z* that censors the negative elements of *x* with zeros.
3 Branching conditions and loops

- **if(condition)**: Tests for condition to be *TRUE*. If it is the case the block is executed.
  
  ```r
  > x <- TRUE
  > if (x) {
  + print("x is TRUE")
  + }
  [1] "x is TRUE"
  ```

- **if (condition) ... else**: *if (condition) tests for condition to be* *TRUE*. If it is the case the first block is executed, otherwise the second block is executed.
  
  ```r
  > x <- FALSE
  > if (x) {
  + print("x is TRUE")
  + } else {
  + print("x is FALSE")
  + }
  [1] "x is FALSE"
  ```

- **while (condition)**: execute the block as long as condition is *TRUE*.
  
  ```r
  > x <- 10
  > while (x > 0) {
  + x <- (x - 3)
  + }
  > x
  [1] -2
  ```

- **for (variable in vector)**: goes through each element of vector (one after the other), assign the current element to variable and execute the block.
  
  ```r
  > x <- c(1, 2, 3)
  > for (i in x) {
  + print(i)
  + }
  [1] 1
  [1] 2
  [1] 3
  ```

  ```r
  > x <- c("a", "b", "c")
  > for (i in x) {
  + print(i)
  + }
  [1] "a"
  [1] "b"
  [1] "c"
  ```
4 Functions

4.1 writing your own functions

A function is made of arguments and a body. The arguments are what is given to the function, the body is the ‘machinery’ of the function. Let’s define a function to calculate the hypothenuse of a triangle:

```r
> hypothenuse <- function(x, y) {
+    z <- sqrt(x^2 + y^2)
+    return(z)
+ }
> hypothenuse(1, 1)
[1] 1.414214
```

A function can return something (i.e. gives back a result). Our function returns a result. We indicate what to return with the function `return()` (surprise …).

The arguments are (x,y). It is possible to define optional arguments (or arguments with a default value). This is achieved in our example by doing something like `function(x, y=2)
```

4.2 editing a function

The function `edit` can be used to edit R objects, including functions. An editor is opened with the `source code` of the function.

```r
hypo.modif <- edit(hypothenuse)
```

The default editor may depends on your installation, but unless exotic settings it should be the `notepad` for Windows and `vi` for Unices. You can specify an alternative editor with the parameter `editor`. Example:

```r
hypo.modif <- edit(hypothenuse, editor="nedit")
```

or with Microsoft Windows:

```r
hypo.modif <- edit(hypothenuse, editor="notepad")
```

To make correction to an existing function, one can use `fix`:

```r
fix(hypothenuse)
```

An alternative practice is to use your favorite text editor as copy/paste your code in the `R` console.

4.3 Miscellaneous useful functions

- `str(object)`. Dump the structure of the object.
- `traceback()`. Give indications about where the last error occured.
- `object.size(object)`. Return an aproximate of the memory used by an object
• gc(). Garbage collector. ‘Clean’ the memory and give the memory usage.
• capabilities(). Tell about the capabilities of the version of R you are using.

4.4 Exercises
• create a function cube: \( \text{cube}(x) = x^3 \)
• create a function factorial: \( \text{fact}(y) = y! \)
• create a function \( f(x, y) = \text{cube}(x) - \text{fact}(y) \)

5 Graphics

R is a powerful tool to generate graphics. Visualization of data can help the analysis.

5.1 plot

The call to \( \text{plot}(\text{object}) \) will create a plot for the object.

\[
> x \leftarrow c(1, 2, -2, 3, 4)
> \text{plot}(x)
\]
The function plot accepts optional parameters (see figure 1). They are too many to be detailed here. We only introduce few of them. The parameter type can be "p" for points, "l" for lines, "b" for both, "s" for steps, "n" for nothing.

The behavior of the function plot changes according to the object(s) it is used with.

5.2 other high-level plots
- barplot(x): barplot of values in x
- hist(x): histogram for the values in x
- pie(x): pie chart of values in x
- pairs(m): matrix of scatter plot of matrix m.
- image(m): image of a matrix m.
- boxplot(x, y, ...): box and whiskers plot of vectors x, y, ...

5.3 low-level graphical functions
- points(x, y): add points to a plot at coordinates (x,y).
- lines(x, y): add lines to a plot, using coordinates (x,y).
- text(x, y, labels): a the labels at the coordinates (x,y).
```r
> par(mfrow = c(2, 2))
> plot(x, y)
> plot(x, y, pch = 1, col = "red")
> plot(x, y, pch = 1, col = c("red", "red", "green", "red"))
> plot(x, y, type = "l")
```

Figure 1: The upper left plot is the result of `plot(x, y)`, upper right plot is the result of `plot(x, y, pch=1, col="red")`, the lower left plot is the result of `plot(x, y, pch=1, col=c("red", "red", "green", "red"))`, and the lower right plot is the result of `plot(x, y, type="l")` (the numbers are added afterwards and correspond to the order of the values in x and y).
5.4 other nifty things
par(mfrow=c(x, y)) splits your plotting device into x rows and y columns.

5.5 printing your plot
R functions with devices. We show below how to print in a postscript device.

\[ \text{dev.print(postscript, file="where/myfile.ps")} \]

5.6 Exercise
- generate a vector \( x \) with 100 random values using the function \( \text{rnorm} \) (mean zero, standard deviation equal to one), then plot an histogram. Repeat the operation few times. You can observe differences (sampling effect).
- generate a vector \( y \) with 100 random values using the function \( \text{rnorm} \) (mean zero, standard deviation equal to one). Make a scatter plot \( x \) vs \( y \), using different 4 different colors (one for \( x < 0, y < 0 \), one for \( x < 0, y > 0 \), one for \( x > 0, y > 0 \) and one for \( x > 0, y < 0 \)).

6 Reading and writing files

6.1 \( R \) scripts
- \( \text{source(filename)} \): read and execute \( R \) code in the file \( filename \).

6.2 data files
- \( \text{read.table(filename)} \): read the data file \( filename \) and store data in a \textbf{data.frame}. A lot of optional parameters can be specified. Useful ones are \texttt{sep} to specify the separator used, \texttt{skip} to skip lines.

7 Packages
Packages of objects can be loaded using the function \texttt{library}. By default the packages \texttt{base} and \texttt{ctest} (classical tests) are loaded.
For example, to load the package \texttt{mva} (MultiVariate Analysis):

\[ \texttt{> library(mva)} \]

Documented topics in a package can be listed with the function \texttt{library}, using the parameter \texttt{help}:

\[ \texttt{> library(help = mva)} \]
8 To know more

- R for Beginners, by Emmanuel Paradis. It can be downloaded from: http://cran.r-project.org/doc/contrib/rdebuts_en.pdf

...and remember to use the help system!

9 vocabulary

You will find here a summary explanation for some of the technical terms. Refer to your programming course (or teacher) for details.

array Table of elements of the same mode. A matrix is an array

character mode for strings of characters.

compilation Transform source code into executable code.

complex mode for complex numbers

data.frame list in which all the elements have the same length.

evironment can thought of as a working space. Objects are in a given environment. The default environment for user defined objects is called .GlobalEnv.

factor mode for factors (i.e. categories).

integer mode for integers

list objects constituted of ordered components of independant mode. Components can be called by position number or name.

logical two values possible TRUE or FALSE.

matrix 2-ways arrays. Elements can be accessed knowing their location (row number and column number)

mode Sometimes called type. The main ones are: integer, numeric, complex, logical, character, factor and list.

numeric mode real numbers.

source code The program (as written by the programmer).

subset extract a subset of an object. [ in general, "[[" to access an element of a list.

type See mode

vector Sequence of elements of the same mode