Applied Statistics and Data Analysis
Lab 1: An introduction to R

Luca Grassetti and Paolo Vidoni
Department of Economics and Statistics, University of Udine

September, 2019

1 Introduction

1.1 Why R

This document contains and incorporates material from *An Introduction to R* by Peter Haschke, available at www.peterhaschke.com/files/IntroToR.pdf.
We use R (www.cran.r-project.org) because it has a number of virtues and advantages compared to other statistical software packages (for example, Stata, SPSS, etc.). More precisely:

- it is open-source and it is free;
- it is cross-platform (Windows, Mac, Linux, Unix);
- it is what many scientists, and also statisticians, use;
- it has a large active, helpful, and friendly user base;
- it is updated regularly and there is a huge amount of additional packages;
- it has unrivaled graphical capabilities;
- it is extremely flexible and it can do or be made to do just about anything.

It is important to note that R is a compiled or interpreted language. This means that all commands and code you give R will be executed immediately. Unlike in uncompiled languages (such as C, Fortran, etc.) you are not required to build a complete program and instead R will interpret things for you. Moreover, it is not a spreadsheet as Excel. So you do not see what is going on. It is not the best tool for non-statistical programming (as, for example, for web scraping). There is an initially somewhat steep learning curve, especially without any programming background.
It is possible to type code directly into the R console or to send code to R via a text editor. Some of the available text editors are:

- Notepad: you can write and save your code, nothing else;
- R Editor: it is automatically get installed with R, it runs on all platforms and it is only a marginal improvement over Notpad;
- Notepad++: it is a really nice lightweight editor for Windows, it is free and open-source and it is highly customizable;
- Emacs: it is probably the editor of choice for an expert programmer, it is cross-platform and it does anything any text editor could ever be asked to do and more; there exist plug-ins making R programming much easier (for example, Emacs Speaks Statistics);
- RStudio: it is not a text editor but a free and cross-platform Integrated Development Environment (IDE) or software application that provides comprehensive facilities especially designed for R programmers, it is extremely powerful and it can be downloaded from www.rstudio.com.

1.2 Using R

Start R by double-clicking on the R desktop icon or on the icon of RStudio. Depending on how you started R, you should see either the R-console or an R session spawned at the command line. If you use RStudio, you see the RStudio console. The symbol `>` means that R is not doing anything and it is just waiting for your input. It is called the prompt.
Some simple R syntax: `#` comments the text, `;` separates commands, `{}` group expressions or commands, `+` is the prompt symbol for uncompleted commands, `-> <- =` assign results of commands to objects.
Type `q()`, or use the graphical menu, to quit R. The sequence of R commands can be saved in the so-called commands history file, with extension .r, .R or .Rhistory with the command `savehistory("history.r")`. To recall your command history use `loadhistory("history.r")`.
The saved history files, as well as specific script files (namely, files with extension .r), can be used to execute groups of commands (with specific shortcuts) or the entire file using the command `source("commands.r")`. Futhermore, the `sink()` function will redirect an R output to a file instead of to the R terminal.
All objects in the current R session (the workspace) can be saved in a file with extension .rda or .Rdata; `save.image()` is just a shortcut to save the current workspace. The workspace is saved in the current working directory with the command `save.image("myfile.Rdata")`. The default directory is that one where R is installed. The current workspace is automatically reloaded the next time R is started. To load a workspace into the current session (if you don't specify the path, the current working directory is assumed), use `load("myfile.Rdata")`.
R gets confused if you use a path in your code like `c:\mydocuments\myfile.txt` (as typically on Windows). This is because R sees `\` as an escape character (its purpose is to start character sequences which have to be interpreted differently from the same characters occurring without the prefixed escape character). Instead, you have to use `c:\\mydocuments\\myfile.txt` or, better, `c:/mydocuments/myfile.txt`, where the slash symbol is considered instead of the backslash. Also on Windows the R path specification will use `/` as path separator.
Some useful commands:

```r
getwd()  # print the current working directory
setwd("c:/docs/mydirectory")  #  change the working directory to mydirectory
ls()  # list the objects in the current workspace
```

Using RStudio it is extremely easy to manage .r or .rnw (Latex files containing R commands) files. Moreover, it is immediate to visualize the content of the working directory.

1.3 R as calculator

R understands the following basic operators:

- `+` and `-` for addition and subtraction;
- $*$ and `/` for multiplication and division;
- `^` for the exponent;
- `%%` is the modulo operator;
- `%\%` for integer division.

Below is an excerpt of some of the basic mathematical functions that R knows:

- `print()` prints objects;
- `log()` computes logarithms;
- `exp()` computes the exponential function;
- `sqrt()` takes the square root;
- `abs()` returns the absolute value;
- `sin()` returns the sine;
- `cos()` returns the cosine;
- `tan()` returns the tangent;
- `asin()` returns the arc-sine;
- `factorial()` returns the factorial;
- `sign()` returns the sign (negative or positive);
- `round()` rounds the input to the desired digit.

As already mentioned an important operator is the comment operator `#`. Whenever R encounters this operator, it will ignore everything printed after it (in the current line). As can be inferred from its name, this is extremely useful to annotate your code. For the most part, R does not care about spacing. Spaces of course matter when you are dealing with character strings. R is case sensitive. As anticipated before, there is another special character, the semicolon `;`. R evaluates code line by line and a line-break tells R that a statement is to be evaluated. Instead of a line-break, you can use a semicolon to tell R where statements end.

```
#  # this is a comment
1+2+3
```

```
[1] 6
```

```
2+3*4
```

```
[1] 14
```

```
3/2+1
```

```
[1] 2.5
```

```
2+(3*4)
```

```
[1] 14
```

```
(2 + 3) * 4
```

```
[1] 20
```

```
4*3^3
```

```
[1] 108
```

```
27^(1/3)
```

```
[1] 3
```

```
2/0 # the result is (positive) infinity
```

```
[1] Inf
```

```
0/0 # the result is Not a Number (NaN)
```

```
[1] NaN
```

```
23%%3
```

```
[1] 2
```

```
23%/%3
```

```
[1] 7
```

```
sqrt(2)
```

```
[1] 1.414214
```

```
sin(3.14159)
```

```
[1] 2.65359e-06
```

```
sin(pi)
```

```
[1] 1.224606e-16
```

1.4 Logical Operators

Among the most used features of R are logical operators. You will use these throughout your code and they are crucial for all sorts of data manipulation. When R evaluates statements containing logical operators it will return either `TRUE` or `FALSE` . Below is a list of most of them.

- `<` less than;
- `<=` less than or equal to;
- `>` greater than;
- `>=` greater than or equal to;
- `==` equal;
- `!=` not equal;
- `&` and;
- `|` or;
- `xor` exclusive disjunction.

In addition to the shorter form `&` and `|`, the conjunction and disjunction operators present the longer form `&&` and `||`. The shorter form performs element-wise comparisons in much the same way as arithmetic operators. The longer form evaluates left to right examining only the first element of each vector. Evaluation proceeds only until the result is determined.

```
1 == 1
```

```
[1] TRUE
```

```
1 == 2
```

```
[1] FALSE
```

```
1 != 2
```

```
[1] TRUE
```

```
1 <= 2 & 1 <= 3
```

```
[1] TRUE
```

```
1 == 1 | 1 == 2
```

```
[1] TRUE
```

```
1 > 1 | 1 > 2 & 3 == 3
```

```
[1] FALSE
```

```
1 > 1 & 1 > 2 & 1 > 3
```

```
[1] FALSE
```

```
xor(TRUE, TRUE)
```

```
[1] FALSE
```

```
xor(TRUE, FALSE)
```

```
[1] TRUE
```

1.5 R's help function

The first thing to do if you have a question about one of R's functions is to ask R. This is important and you will use this functionality a lot. For example, we can simply pull up a help page for the `lm()` function with the following commands

```
?lm
help(lm)
# ??lm search for every function related to lm (the pattern may be prefixed with
```

```
# a package name followed by :: or ::: to limit the search to that package)
```

If the name of the function is not exactly known, you can use the `apropos()` function. It will return a list of all functions known to R containing the search term.

```
apropos("mean")  # when the name of the function is not known exactly
```

A help page will appear with the following main sections:

- Description: purpose of the function;
- Usage: an example of a typical implementation;
- Arguments: a list of the arguments you can supply to the function and what each does;
- Details: more detailed information about the function and its arguments;
- Value: information about the likely output of a function (for example, the function may return an integer, or a list, or a matrix, or something different);
- See Also: a list of useful related functions;
- References: citations which can often be very useful;
- Examples: example code.

If you cannot find your answer with the above methods, the best place to probe further is the overall R help feature via the `help.start()` function (it starts the html help version). This opens a comprehensive helping of R documentation, manuals, and help for all installed packages in your web-browser.

1.6 Packages and libraries

The base version of R contains a number of packages. But there are hundreds of packages to install (see, for example, the CRAN web page). To install a package you have to run the following command `install.packages("package")`.
You will be asked to pick a CRAN mirror from which to download (generally the closer the faster) and R will install the package to your library. R will still be clueless. To actually tell R to use the new package you have to tell R to load the package's library each time you start an R session, using the command `library("package")`. Packages are frequently updated. Depending on the developer this could happen very often. To keep your packages updated enter this every once in a while: `update.packages()`.

2 The building blocks

To move beyond using R as a calculator this chapter will introduce the main building blocks of R: objects and their modes.

2.1 Objects, assignment and reference

R is an object oriented language. Everything in R is an object. When R does anything, it creates and manipulates objects. R's objects come in different types and flavors. The most basic ones are:

- vectors: these are one-dimensional sequences of elements of the same type, that is real (double), integer, logical or character elements;
- matrices and arrays: these are two dimensional rectangular objects (matrices) and higher-dimensional rectangular objects (arrays); all elements of matrices or arrays have to be of the same type;
- lists: lists are like vectors but they do not have to contain elements of the same type; the first element of a list could be a vector of the 26 letters of the alphabet, the second element could contain a vector of all the prime numbers below 1000 and the third could be a 2 by 7 matrix;
- data frames: data frames are best understood as special matrices (technically they are a type of list); for most applications involving data sets you will use data frames and they are two dimensional containers with rows corresponding to *observations* and columns corresponding to *variables*;
- factors: factors are vectors that can contain only predefined values, and they are used to store categorical data; they behave differently than vectors;
- functions: functions are objects that take other objects as inputs and return some new object as output.

All objects have a certain mode and a certain type. Some objects can only deal with one type of elements at a time, others can store elements of multiple types. R distinguishes the following main types:

- integer: integer numbers (numeric mode);
- real (double): real numbers (numeric mode);
- complex: complex or imaginary numbers (complex mode);
- character: elements made up of text-strings (character mode); for example, `"text"`, `"Hello!"`, or `"123"`;
- logical: data containing logical constants (logical mode), namely, `TRUE` and `FALSE`.

Knowing the types of objects R can work with is not terribly useful without knowing how to store these objects and without knowing how to recall or reference them when needed. You often will compute some statistic or manipulate some matrix. Instead of recalculating everything over and over again we can give things names and recall them later. Below we will cover how to create, assign to and refer to various objects.
Recall our basic arithmetic examples from above. We implicitly relied on and then manipulated objects and R implicitly printed these objects to the screen. The result is not saved, it is only printed on the screen.

```
1 + 2

[1] 3
```

Let's assign and recall names instead. We can do that by using the assignment operator `<-` or `=`.

```
x <- sqrt(2)   # the square root of 2 is saved in x
x   # in order to print the content of x

[1] 1.414214

x^3

[1] 2.828427

y <- x^3
y

[1] 2.828427
```

A new assignment erases the former one.

```
x <- pi
x

[1] 3.141593
```

The `is()` function, when supplied with the name of an object, will tell you what type of object we have as well as its mode.
To see what objects are in the current working directory use the command `ls()`. If you want to remove an object from memory use the `rm()` function, supplied with the name of the object. This will delete the object permanently. If you want to remove all objects from active memory, this will do the trick `rm(list = ls())`.
R stores data in the following data structures already mentioned before: vector, list, matrix, array, factor and data frame.

2.2 Vectors

In order to create a vector you will use the `c()` function. The `c` stands for concatenate, and you can string a bunch of elements together, separated by commas.

```
Vector1 <- c(1,2,3,4,5,6,7,8,9,10) # numerical vector
Vector1

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

```
x <- c(2,3,5,7)
x

[1] 2 3 5 7

x <- c(x,11)
x

[1]  2  3  5  7 11
```

It is possible to create character vectors; characters are indicated in quotation marks (inverted commas).

```
Vector2 <- c("a","b","c","d") # character vector
Vector2

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

Vector3 <- c("1","2","3","4")  # character vector (numbers viewed as characters)
Vector3

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

Logical vectors are vectors of logical elements.

```
x = c(TRUE, FALSE, TRUE, FALSE)
y = !x
x

[1]  TRUE FALSE  TRUE FALSE

y

[1] FALSE  TRUE FALSE  TRUE

x & y

[1] FALSE FALSE FALSE FALSE

x | y

[1] TRUE TRUE TRUE TRUE
```

You can also string multiple vectors together with the `c()` function.

```
Vector4 <- c(Vector2 , Vector3 , Vector2 , Vector2 , Vector2)
Vector4

 [1] "a" "b" "c" "d" "1" "2" "3" "4" "a" "b" "c" "d" "a" "b" "c" "d"
[17] "a" "b" "c" "d"
```

Most of the time, using the `c()` function will be tedious as you do not want to manually type all elements of a vector. Luckily, there are a number of alternative commands. In particular, you can use the colon `:` to tell R to create an integer vector with a regular numbers sequence.

```
xx <- 1:10
xx

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

5:-5

 [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

Or you may use the `seq()` function, which is more general and has some neat features.

```
seq(from=0,to=10) # you can drop the argument names

 [1]  0  1  2  3  4  5  6  7  8  9 10

seq(0,10)

 [1]  0  1  2  3  4  5  6  7  8  9 10

seq(0,10,by = 2) # the by argument let's you set the increments

[1]  0  2  4  6  8 10

seq(0,10,length.out=25)

 [1]  0.0000000  0.4166667  0.8333333  1.2500000  1.6666667  2.0833333
 [7]  2.5000000  2.9166667  3.3333333  3.7500000  4.1666667  4.5833333
[13]  5.0000000  5.4166667  5.8333333  6.2500000  6.6666667  7.0833333
[19]  7.5000000  7.9166667  8.3333333  8.7500000  9.1666667  9.5833333
[25] 10.0000000

# the length.out argument specifies the length of the vector and
# R figures out the increments itself
```

Another useful function is `rep()` which allows you to repeat things.

```
rep(0,time=10)

 [1] 0 0 0 0 0 0 0 0 0 0

rep("Hello",3) # you can drop the argument name

[1] "Hello" "Hello" "Hello"

rep(Vector1,2)

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

rep(Vector2,each=2) # we can repeat each element as well

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

2.3 Vector operations

Most standard mathematical functions work with real vectors. The basic operators, the logical operators and the mathematical functions are applied element-wise.

- `sum()` sums of the elements of the vector;
- `prod()` product of the elements of the vector;
- `min()` minimum of the elements of the vector;
- `max()` maximum of the elements of the vector;
- `mean()` mean of the elements;
- `median()` median of the elements;
- `range()` the range of the vector;
- `var()` the variance (division by `n-1`);
- `sd()` the standard deviation;
- `cov()` the covariance (takes two inputs, for example `cov(x,y)`);
- `cor()` the correlation coefficient (takes two inputs, for example `cov(x,y)`);
- `sort()` sorts the vector elements (default argument `decreasing = FALSE`)
- `order()` indices of the vector elements in ascending order;
- `length()` returns the length of the vector;

- `summary()` returns summary statistics;
- `which()` returns the index after evaluating a logical statement;
- `which.min()` returns the index of the min;
- `which.max()` returns the index of the max;
- `unique()` returns a vector of all the unique elements of the input;
- `round()` rounds the values in its first argument to the specified number of decimal places (the default is 0).

```
x <- 0:10
x+1
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11
```

```
y <- -5:5
abs(y)
```

```
 [1] 5 4 3 2 1 0 1 2 3 4 5
```

```
x+y
```

```
 [1] -5 -3 -1  1  3  5  7  9 11 13 15
```

```
x*y
```

```
 [1]  0 -4 -6 -6 -4  0  6 14 24 36 50
```

```
y <- 0:8
x+y
```

```
Warning in x + y:  longer object length is not a multiple of shorter object length
```

```
 [1]  0  2  4  6  8 10 12 14 16  9 11
```

When two objects have different lengths, R reuses elements of the shorter one.

```
x
```

```
 [1]  0  1  2  3  4  5  6  7  8  9 10
```

```
x > 5
```

```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```r
x <- 3:26
max(x)
```

```
[1] 26
```

```r
min(x)
```

```
[1] 3
```

```r
sum(x)
```

```
[1] 348
```

```r
prod(x)
```

```
[1] 2.016457e+26
```

```r
x <- c(32,18,25:21,40,17)
x
```

```
[1] 32 18 25 24 23 22 21 40 17
```

```r
sort(x)  # ascending order
```

```
[1] 17 18 21 22 23 24 25 32 40
```

```r
order(x) # position indices of the elements in ascending order
```

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

```r
Vector1+Vector1
```

```
 [1]  2  4  6  8 10 12 14 16 18 20
```

```r
Vector1/Vector1
```

```
 [1] 1 1 1 1 1 1 1 1 1 1
```

```r
log(Vector1)
```

```
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595
 [7] 1.9459101 2.0794415 2.1972246 2.3025851
```

```r
round(log(Vector1))
```

```
 [1] 0 1 1 1 2 2 2 2 2 2
```

```r
round(log(Vector1),digit = 3)
```

```
 [1] 0.000 0.693 1.099 1.386 1.609 1.792 1.946 2.079 2.197 2.303
```

```r
which(Vector1>=5) # this returns the indices not the elements
```

```
[1]  5  6  7  8  9 10
```

2.4 Vector indexing

Sometimes you do not want to print or manipulate an entire vector. This is where indexing comes in. Indexing vectors is done with square brackets `[ ]`.

```
Vector6 <- c("The","Starlab","Fellow","is","a Fool")
Vector6 [3]

[1] "Fellow"

Vector6[2:4]

[1] "Starlab" "Fellow"  "is"

Vector6[c(1 ,3 ,4)]

[1] "The"    "Fellow" "is"

Vector6[-2] # all except the 2nd element

[1] "The"    "Fellow" "is"     "a Fool"

Vector6[5] <- "great"
Vector6

[1] "The"     "Starlab" "Fellow"  "is"      "great"

xx <- 100:1
xx[7]

[1] 94

xx[c(2,3,5,7,11)]

[1] 99 98 96 94 90

xx[85:91]

[1] 16 15 14 13 12 11 10

xx[91:85]

[1] 10 11 12 13 14 15 16

xx[c(1:5,8:10)]

[1] 100  99  98  97  96  93  92  91

xx[c(1,1,1,1,2,2,2,2)]
```

```
[1] 100 100 100 100  99  99  99  99
```

```
yy <- xx[c(1,2,4,8,16,32,64)]
yy
```

```
[1] 100  99  97  93  85  69  37
```

```
x <- c(32,18,25:21,40,17)
x
```

```
[1] 32 18 25 24 23 22 21 40 17
```

```
sort(x)
```

```
[1] 17 18 21 22 23 24 25 32 40
```

```
x[order(x)]
```

```
[1] 17 18 21 22 23 24 25 32 40
```

```
x <- c(1,2,4,8,16,32)
x
```

```
[1]  1  2  4  8 16 32
```

```
x[-4]
```

```
[1]  1  2  4 16 32
```

```
x[-c(3,4)]
```

```
[1]  1  2 16 32
```

Logical operators come in handy when indexing.

```
x <- -8:7
x<0
```

```
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE FALSE
```

```
x[x<0]
```

```
[1] -8 -7 -6 -5 -4 -3 -2 -1
```

```
x[x<0&x<(-2)]
```

```
[1] -8 -7 -6 -5 -4 -3

x[x<0|x>5]

 [1] -8 -7 -6 -5 -4 -3 -2 -1  6  7

x[x!=6]

 [1] -8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  7

x[x==6]

[1] 6
```

2.5 More functions

The `na.omit()` function returns the vector suppressing the `NA`s (`NA` = not available, the missing value indicator) and adds an attribute to it called `na.action`. This is helpful because now we can compute all those functions that break when they encounter `NA`s.

```
z<-c(2,3,4,3,NA,NA,6,6,10,11,2,NA,4,3)
max(z) # this won't work because many function can't deal with NAs

[1] NA

na.omit(z) # returns the vector supressing the Nas

 [1]  2  3  4  3  6  6 10 11  2  4  3
attr(,"na.action")
[1]  5  6 12
attr(,"class")
[1] "omit"

max(na.omit(z))

[1] 11

max(z,na.rm=T)

[1] 11
```

As you can see, an alternative is to consider the `na.rm=TRUE` option.
The `is.na()` function indicates which elements are missing. With the `subset()` function we can remove the `NA`s manually.

```
is.na(z)

 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
[12]  TRUE FALSE FALSE

z.noNA <- subset(z,is.na(z)==FALSE )
z.noNA

 [1]  2  3  4  3  6  6 10 11  2  4  3
```

The `subset()` function can be used for more than `NA` removal. Let us use it to find numbers divisible by 7.

```
X <- 1:70
Multiple7 <- subset(X,X%%7==0) # recall the modulo operator
Multiple7

 [1]  7 14 21 28 35 42 49 56 63 70
```

2.6 Ordered and unordered factors

Factors are vectors that can contain only predefined values, and they are used to store categorical data. Since categorical variables enter into statistical models differently than numeric variables, storing data as factors insures that the modeling functions will treat such data correctly. Factors in R are stored as a vector of integer values with a corresponding set of character values to use when the factor is displayed. The `factor()` function is used to create a factor. The only required argument to factor is a vector of values which will be returned as a vector of factor values. Both numeric and character variables can be made into factors, but a factor's levels will always be character values. You can see the possible levels for a factor through the `levels()` command.
Thus a factor is a character vector object used, for example, to specify a discrete classification (grouping) of the components of other vectors of the same length. R provides both ordered and unordered factors.
Suppose that 6 subjects are observed: two of them received treatment `a`, three treatment `b` and one treatment `c`. To save this information a factor is created. To find out the levels of a factor the function `levels()` can be used.

```
treat <- factor(c("a","b","b","c","a","b"))
treat

[1] a b b c a b
Levels: a b c

levels(treat)

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

Supposed that for the 6 subjects the following response are observed.

```
resp <- c(10,3,7,6,4,5)
resp[treat=="a"]

[1] 10  4

sum(resp[treat=="b"])

[1] 15
```

The levels of factors are stored in alphabetical order, or in the order they were specified to factor if they were specified explicitly. Sometimes the levels will have a natural ordering that we want to record and we want our statistical analysis to make use of. The `ordered()` function creates such ordered factors but is otherwise identical to factor. For most purposes the only difference between ordered and unordered factors is that the former are printed showing the ordering of the levels.

```
treat1 <- ordered(c("a","b","b","c","a","b"), levels=c("c","b","a")) # to set a
# different ordering
treat1

[1] a b b c a b
Levels: c < b < a

levels(treat1)

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

It is possible to create a factor by grouping numerical observations in classes.

```
x<-c(1:12,25:38,-3:0,13:24)
x1 <- cut(x,c(-5,5,25,40),labels=c("B","M","A"))
# classes: (-5,5],(5,25],(25,40] corresponding to B, M, A
x1

 [1] B B B B B M M M M M M M M A A A A A A A A A A A A A B B B B M M M
[34] M M M M M M M M M
Levels: B M A

levels(x1)

[1] "B" "M" "A"

f_x1 <- table(x1) # frequencies for each level
f_x1

x1
 B  M  A
 9 20 13
```

It is possible to apply the same function to values in a numerical vector, which are grouped by a particular feature.

```
x<-1:20
y<-factor(rep(0:1,10))
y

 [1] 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Levels: 0 1

tapply(x,y,sum)

  0   1
100 110

# apply sum to each group of values in x given by a unique combination of the
# levels of the factor y
```

Factors represent a very efficient way to store character values, because each unique character value (level) is stored only once, and the data itself is stored as a vector of integers. Because of this, most data loading functions in R automatically convert character vectors into factors. In order to avoid this, it is necessary to define a specific argument in the loading functions.

2.7 Arrays and matrices

An array can be considered as a multiply subscripted collection of data entries of the same type. R allows simple facilities for creating and handling arrays, and in particular the special case of matrices (two-dimensional arrays). Here we focus on matrices.
To create matrices we will use the `matrix()` function which takes the following arguments:

- data: an R object (it could be a vector);
- nrow: the desired number of rows;
- ncol: the desired number of columns;
- byrow: a logical statement to populate the matrix by either row or column.

```
Matrix1 <- matrix(data=1,nrow=3,ncol=3)
Matrix1

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

```
dim(Matrix1)

[1] 3 3

# matrix filled with a vector of values
Vector8 <- 1:12
Vector8

 [1]  1  2  3  4  5  6  7  8  9 10 11 12

Matrix3 <- matrix(data=Vector8,nrow=4)
# by default the matrix will be populated by column
Matrix3

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

Matrix4 <- matrix(data=Vector8,nrow=4,byrow=TRUE)
# now we populated it by row
Matrix4

     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
```

You can also create matrices by pasting together vectors using the `rbind()` (pasting vectors by rows) and `cbind()` (pasting vectors by columns) functions. The `rbind()` and `cbind()` functions automatically label row or column names. You can use the `rownames()` and `colnames()` functions to manipulate these.

```
Vector9 <- 1:10
Vector9

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

Vector10 <- Vector9^2
Vector10

 [1]   1   4   9  16  25  36  49  64  81 100

Matrix5 <- rbind(Vector9,Vector10)
Matrix5
```

```
         [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
Vector9     1    2    3    4    5    6    7    8    9    10
Vector10    1    4    9   16   25   36   49   64   81   100
```

```
Matrix6 <- cbind(Vector9,Vector10,Vector9)
Matrix6
```

```
      Vector9 Vector10 Vector9
 [1,]       1        1       1
 [2,]       2        4       2
 [3,]       3        9       3
 [4,]       4       16       4
 [5,]       5       25       5
 [6,]       6       36       6
 [7,]       7       49       7
 [8,]       8       64       8
 [9,]       9       81       9
[10,]      10      100      10
```

```
colnames(Matrix6)
```

```
[1] "Vector9"  "Vector10" "Vector9"
```

```
rownames(Matrix6) # the rownames do not exist
```

```
NULL
```

```
colnames(Matrix6) <- c("A","B","C")
rownames(Matrix6) <- c("a","b","c","d","e","f","g","h","i","j")
Matrix6
```

```
   A   B  C
a  1   1  1
b  2   4  2
c  3   9  3
d  4  16  4
e  5  25  5
f  6  36  6
g  7  49  7
h  8  64  8
i  9  81  9
j 10 100 10
```

The `diag()` function is useful for creating a diagonal matrix.

```
Matrix7 <- diag(5) # creates a 5 by 5 identity matrix
Matrix7

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1

Vector11 <- c(1,2,3,4,5)
Matrix8 <- diag(Vector11) # Vector11 across the diagonal
Matrix8

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    2    0    0    0
[3,]    0    0    3    0    0
[4,]    0    0    0    4    0
[5,]    0    0    0    0    5

diag(Matrix7) # extracts the diagonal from a matrix

[1] 1 1 1 1 1
```

R can do matrix arithmetic. Below is a list of some basic operations we can do.

- `+ - * /` standard scalar or by element operations;
- `%*%` matrix multiplication;
- `t()` transpose;
- `solve()` inverse;
- `det()` determinant;
- `chol()` cholesky decomposition;
- `eigen()` eigenvalues and eigenvectors;
- `crossprod()` cross product;
- `%x%` kronecker product.

2.8 Indexing matrices

Matrices are indexed via `[ ]`, as done for vectors. Using `[i,j]` will retrieve the j-th element of the i-th row. You can also extract entire rows as vectors by leaving the column entry blank, and similarly for extracting an entire column.

```
Matrix9 <- matrix(1:9,3)
Matrix9

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

Matrix9[1,1] # extracts the first element of the first row

[1] 1

Matrix9[2,3] # extracts the third element of the second row

[1] 8

Matrix9[,1] # extracts the first column

[1] 1 2 3

Matrix9[2,] # extracts the second row

[1] 2 5 8

Matrix9[1:2 , ] # extracts the first and the second rows

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

Matrix9[Matrix9[,2]>4,]

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

# extracts all rows that in their second column contain values greater than four
```

Function `apply(x,margin,function)` returns a vector, an array or a list of values obtained by applying a function `function` to margins `margin` of an array or matrix `x`. In case of matrices, `margin=1` corresponds to rows and `margin=2` corresponds to columns.

```
x <- matrix(1:16,ncol=4)
x

     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16

apply(x,1,sum) # applying a function to rows (margin=1)

[1] 28 32 36 40

y<-apply(x,2,prod) # applying a function to columns (margin=2)
y

[1]    24  1680 11880 43680
```

Function `outer(x,y,function)`, when applied to numerical vectors `x` and `y`, returns a matrix with entries obtained by applying a function `function` (the default is the product) to all the combinations of the elements of `x` and `y`.

```
x<-1:5
y<-1:5
outer(x,y)

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25

# returns a matrix with entries obtained by applying a function (the default
# is the product) to all the combinations of the elements of x and y
outer(x,y,"+")

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

# returns a matrix with entries obtained by applying the function sum to all
# the combinations of the elements of x and y
```

2.9 Lists

An R list is an object consisting of an ordered collection of objects known as its components. There is no particular need for the components to be of the same mode or type, and, for example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, a function, and so on. A list can be created using the command `list()`.

```
Lst <- list(name="Fred", wife="Mary", no.children=3,child.ages=c(4,7,9))
Lst

$name
[1] "Fred"

$wife
[1] "Mary"

$no.children
[1] 3

$child.ages
[1] 4 7 9

Lst[[1]]

[1] "Fred"

Lst[[4]]

[1] 4 7 9

Lst$child.ages

[1] 4 7 9

Lst$child.ages[2]

[1] 7
```

Components are always numbered and may always be referred to as such. Thus if `Lst` is the name of a list with four components, these may be individually referred to as `Lst[[1]]`, `Lst[[2]]`, `Lst[[3]]` and `Lst[[4]]`. If, further, `Lst[[4]]` is a vector subscripted array then `Lst[[4]][1]` is its first entry. The function `length(Lst)` gives the number of (top level) components of the list. Components of lists may also be named, and in this case the component may be referred to either by giving the component name as a character string in place of the number in double square brackets, or, more conveniently, by giving an expression of the form `list_name$component.name`.

2.10 Data frame

A data frame is a list with class `data.frame`. There are restrictions on lists that may be made into data frames, namely:

- the components must be vectors (numeric, character, or logical), factors, numeric matrices, lists, or other data frames;
- matrices, lists, and data frames provide as many variables to the new data frame as they have columns, elements, or variables, respectively;
- numeric vectors, logical vectors and factors are included as elements, and by default character vectors are coerced to be factors, whose levels are the unique values appearing in the vector;
- vector structures appearing as variables of the data frame must all have the same length, and matrix structures must all have the same row size.

Therefore, a data frame is a list of named vectors (called columns) of the same length. A data frame is like a database table. Hence different columns have the same length but can have different types. Each column has a name and contains elements of the same type. Like a list, elements (columns) can have different types. Like a matrix, columns have the same length.
A data frame may, for many purposes, be regarded as a matrix with columns possibly of differing types and attributes. It may be displayed in matrix form, and its rows and columns extracted using matrix indexing conventions.
Whenever you encounter data sets they will usually be stored in data frame objects, which is the most flexible and, arguably, the most used object type. It can be interpreted as a data matrix, where the rows correspond to statistical units and the columns to the interest variables.
The easiest way to load a data set is with the `data()` function. If entered without arguments, it will bring up a list of all data sets that come bundled with R. To save a copy of the data set you can use the `save()` function.
The data set `airquality`, available in R, contains daily air quality measurements in New York from May to September 1973. It is a data frame with 154 observations on 6 variables.

```
data(airquality) # data set, available in R, with daily air quality measurements
# in New York from May to September 1973
dim(airquality) # data frame with 154 observations on 6 variables

[1] 153   6

help(airquality)  #  description of the data set

starting httpd help server ...  done

names(airquality)  # gives all the variable names of the data frame

[1] "Ozone"   "Solar.R" "Wind"    "Temp"    "Month"   "Day"

head(airquality)
```

```
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6

# returns the first 6 rows of the data frame, alternatively
# use airquality[1:6,]
```

Function `str()` summarizes the internal structure of an R object, and in particular of a data set.

```
str(airquality) # summarized the internal structure of an R object

'data.frame': 153 obs. of  6 variables:
 $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
 $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

Most of the basic extraction principles we used for matrices (namely the `[ ]`) also work for data frames. But you should remember that data frames are a special type of list and as such the `$` will come in handy. You can combine the `$` notation and the `[ ]` notation.

```
airquality$Ozone

  [1]  41  36  12  18  NA  28  23  19   8  NA   7  16  11  14  18  14
 [17]  34   6  30  11   1  11   4  32  NA  NA  NA  23  45 115  37  NA
 [33]  NA  NA  NA  NA  NA  29  NA  71  39  NA  NA  23  NA  NA  21  37
 [49]  20  12  13  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA 135  49  32
 [65]  NA  64  40  77  97  97  85  NA  10  27  NA   7  48  35  61  79
 [81]  63  16  NA  NA  80 108  20  52  82  50  64  59  39   9  16  78
 [97]  35  66 122  89 110  NA  NA  44  28  65  NA  22  59  23  31  44
[113]  21   9  NA  45 168  73  NA  76 118  84  85  96  78  73  91  47
[129]  32  20  23  21  24  44  21  28   9  13  46  18  13  24  16  13
[145]  23  36   7  14  30  NA  14  18  20

airquality$Ozone[1:5]

[1] 41 36 12 18 NA
```

The problem with data sets being lists is that working with the `$` notation is kind of tedious. Luckily there exists a function that makes dealing data frame names easier. Whenever you need to

extract or index multiple variables and do not feel like typing `dataset$variable.name` each time, use the `with()` function.

```
airquality$Ozone + airquality$Wind/airquality$Temp
```

```
  [1]  41.110448  36.111111  12.170270  18.185484         NA
  [6]  28.225758  23.132308  19.233898   8.329508         NA
 [11]   7.093243  16.140580  11.139394  14.160294  18.227586
 [16]  14.179688  34.181818   6.322807  30.169118  11.156452
 [21]   1.164407  11.227397   4.159016  32.196721         NA
 [26]         NA         NA  23.179104  45.183951 115.072152
 [31]  37.097368         NA         NA         NA         NA
 [36]         NA         NA  29.118293         NA  71.153333
 [41]  39.132184         NA         NA  23.097561         NA
 [46]         NA  21.193506  37.287500  20.141538  12.157534
 [51]  13.135526         NA         NA         NA         NA
 [56]         NA         NA         NA         NA         NA
 [61]         NA 135.048810  49.108235  32.113580         NA
 [66]  64.055422  40.131325  77.057955  97.068478  97.061957
 [71]  85.083146         NA  10.195890  27.183951         NA
 [76]   7.178750  48.085185  35.125610  61.075000  79.058621
 [81]  63.135294  16.093243         NA         NA  80.100000
 [86] 108.094118  20.104878  52.139535  82.084091  50.086047
 [91]  64.089157  59.113580  39.085185   9.170370  16.090244
 [96]  78.080233  35.087059  66.052874 122.044944  89.114444
[101] 110.088889         NA         NA  44.133721  28.140244
[106]  65.121250         NA  22.133766  59.079747  23.097368
[111]  31.139744  44.132051  21.201299   9.198611         NA
[116]  45.122785 168.041975  73.093023         NA  76.100000
[121] 118.024468  84.065625  85.067021  96.075824  78.055435
[126]  73.030108  91.049462  47.085057  32.184524  20.136250
[131]  23.132051  21.145333  24.132877  44.183951  21.203947
[136]  28.081818   9.153521  13.161972  46.088462  18.205970
[141]  13.135526  24.151471  16.097561  13.196875  23.129577
[146]  36.127160   7.149275  14.263492  30.098571         NA
[151]  14.190667  18.105263  20.169118
```

```
with(airquality, Ozone + Wind/Temp)  # gives the same result as the above command
```

```
  [1]  41.110448  36.111111  12.170270  18.185484         NA
  [6]  28.225758  23.132308  19.233898   8.329508         NA
 [11]   7.093243  16.140580  11.139394  14.160294  18.227586
 [16]  14.179688  34.181818   6.322807  30.169118  11.156452
 [21]   1.164407  11.227397   4.159016  32.196721         NA
 [26]         NA         NA  23.179104  45.183951 115.072152
```

```
 [31]  37.097368         NA         NA         NA         NA
 [36]         NA         NA  29.118293         NA  71.153333
 [41]  39.132184         NA         NA  23.097561         NA
 [46]         NA  21.193506  37.287500  20.141538  12.157534
 [51]  13.135526         NA         NA         NA         NA
 [56]         NA         NA         NA         NA         NA
 [61]         NA 135.048810  49.108235  32.113580         NA
 [66]  64.055422  40.131325  77.057955  97.068478  97.061957
 [71]  85.083146         NA  10.195890  27.183951         NA
 [76]   7.178750  48.085185  35.125610  61.075000  79.058621
 [81]  63.135294  16.093243         NA         NA  80.100000
 [86] 108.094118  20.104878  52.139535  82.084091  50.086047
 [91]  64.089157  59.113580  39.085185   9.170370  16.090244
 [96]  78.080233  35.087059  66.052874 122.044944  89.114444
[101] 110.088889         NA         NA  44.133721  28.140244
[106]  65.121250         NA  22.133766  59.079747  23.097368
[111]  31.139744  44.132051  21.201299   9.198611         NA
[116]  45.122785 168.041975  73.093023         NA  76.100000
[121] 118.024468  84.065625  85.067021  96.075824  78.055435
[126]  73.030108  91.049462  47.085057  32.184524  20.136250
[131]  23.132051  21.145333  24.132877  44.183951  21.203947
[136]  28.081818   9.153521  13.161972  46.088462  18.205970
[141]  13.135526  24.151471  16.097561  13.196875  23.129577
[146]  36.127160   7.149275  14.263492  30.098571         NA
[151]  14.190667  18.105263  20.169118
```

The `subset()` function works the same way for data sets.

```
subset(airquality, airquality$Month == 5 & airquality$Solar.R >= 150)
```

```
   Ozone Solar.R Wind Temp Month Day
1     41     190  7.4   67     5   1
4     18     313 11.5   62     5   4
7     23     299  8.6   65     5   7
10    NA     194  8.6   69     5  10
12    16     256  9.7   69     5  12
13    11     290  9.2   66     5  13
14    14     274 10.9   68     5  14
16    14     334 11.5   64     5  16
17    34     307 12.0   66     5  17
19    30     322 11.5   68     5  19
22    11     320 16.6   73     5  22
26    NA     266 14.9   58     5  26
29    45     252 14.9   81     5  29
```

```
30   115     223  5.7   79     5  30
31    37     279  7.4   76     5  31
```

R is not very good for editing data sets or data entry generally. This is not surprising since R is not a spreadsheet. A data frame can be created using the command `data.frame()`.

```
x<-1:5
y<-factor(c("a","b","a","a","b"))
z<-matrix(rep(7,15),nrow=5,byrow=F)
es.df<-data.frame(z,uno=x,due=y)
es.df

  X1 X2 X3 uno due
1  7  7  7   1   a
2  7  7  7   2   b
3  7  7  7   3   a
4  7  7  7   4   a
5  7  7  7   5   b
```

A matrix can also be transformed into a data frame by considering the `as.data.frame()` command.

```
z<-matrix(rep(7,15),nrow=5,byrow=F) # a matrix transformed into a data frame
z

     [,1] [,2] [,3]
[1,]    7    7    7
[2,]    7    7    7
[3,]    7    7    7
[4,]    7    7    7
[5,]    7    7    7

class(z)

[1] "matrix"

z_df<- as.data.frame(z)
z_df

  V1 V2 V3
1  7  7  7
2  7  7  7
3  7  7  7
4  7  7  7
5  7  7  7

class(z_df)

[1] "data.frame"
```

2.11 Loading and saving data sets

In virtually all cases, you will not find the data you want or need bundled in R. Frequently, you will have created or downloaded data in some other program (for example, Excel, Stata, or simply a text file).
Large data objects will usually be read as values from external files rather than entered during an R session at the keyboard. R input facilities are simple and their requirements are fairly strict and even rather inflexible. There is a clear presumption by the designers of R that you will be able to modify your input files using other tools, such as file editors to fit in with the requirements of R.
If variables are to be held mainly in data frames, as we strongly suggest they should be, an entire data frame can be read directly with the `read.table()` function. There is also a more primitive input function, `scan()`, that can be called directly.
To read an entire data frame directly, the external file will normally have a special form. The first line of the file should have a name for each variable in the data frame. Each additional line of the file has as its first item a row label and the values for each variable.
By default numeric items (except row labels) are read as numeric variables and non-numeric variables are read as factors. This can be changed if necessary. In particular, since R automatically convert character vectors to factors, it is possible to use the argument `stringsAsFactors=FALSE` to suppress this. Another possibility is to load non-numeric variables as factors and to convert some columns to characters.
The function `read.table()` is a general function which can be used to read the data frame directly and it allows to set the delimiter, to include or not the headers, and more. In order to read data from a text file it is possible to use the following commands.

```
dat <- read.table("file.txt", sep=" ", header=T)
dat <- read.table("file.cvs", sep=",", header=T)
```

The `header=TRUE` option specifies that the first line is a line of headings, and hence, by implication from the form of the file, that no explicit row labels are given. The field separator character is defined by `sep`. If `sep = " "` (the default for `read.table`) the separator is the *white space*; `sep = ","` sets a comma separator value, as for data in `.csv` format, which can be created, for example, using Excel (the function `read.csv()` considers the comma as default separator value).
It is also possible to load data created in some other program such as Excel (for example using function `read_xlsx()` of the package `readxl`), SPSS (using function `read.spss()` of the package `foreign`), STATA (using function `read.dta()` of the of the package `foreign`) or to load a data set from a website or to import data through web scraping.
When the data has fixed-width columns separated by spaces, we can simply use `read.table()` with `strip.white=TRUE`, which remove extra spaces. Moreover, if the data file has columns containing spaces, or columns with no spaces separating them, we may use function `read.fwf()`, which allows also the specification of the columns width.
The data set `airquality` is now modified in order to exclude the rows where the variable `Ozone` is `NA`.

```
airq<-airquality[!is.na(airquality$Ozone),]
```

In order to save this new data frame in a text file (in the current working directory), we can use the general function `write.table()` (or the function `write.csv()`, which considers straight away a comma separator value).

```
# to save data with white space as separator
# row.names=F assures that the row names are not written along airq
write.table(airq,file="airq.txt",sep=" ",row.names=F)
# to save with comma as separator
write.table(airq,file="airq.csv",sep=",",row.names=F)
```

In order to load the new data frame.

```
airq1<-read.table("airq.txt",header=T,sep=" ")
airq2<-read.table("airq.csv",header=T,sep=",")
```

The option `row.names=F` in `write.table()` assures that the row names of `airq` are not written along `airq.txt` or `airq.csv`. Thus, the uploaded data frames `airq1` and `airq2` presents row names which do not account the fact that some rows of the former data frame `airquality` are deleted.
The `$` notation, such as in `airquality$Ozone`, for data frame or list components is not always very convenient. A useful facility would be somehow to make the components of a list or data frame temporarily visible as variables under their component name, without the need to quote the list or data frame name explicitly each time. The `attach()` function takes a database such as a list or a data frame as its argument. To detach a data frame, use the function `detach()`.

```
attach(airquality)
Ozone[1:3]

[1] 41 36 12

detach(airquality)
Ozone[1:3]

Error in eval(expr, envir, enclos):  oggetto "Ozone" non trovato
```

2.12 Other useful functions

To determine what type of object and what mode we are dealing with, the `class()`, `mode()`, `typeoff()` and `is()` functions can be employed.
The system can also be queried in order to explore the objects characteristics. The following commands can be used to this end (the result is a logical value):

- `is.numeric()`
- `is.vector()`
- `is.factor()`
- `is.matrix()`
- `is.data.frame()`
- `is.character()`

The following commands can be used to transform R objects:

- `as.numeric()` turns vectors, and matrices of other modes into a numeric ones;
- `as.character()` turns vectors, and matrices of other modes/types into character ones;
- `as.integer()` turns vectors, and matrices of other types into integer ones;
- `as.factor()` turns a vector, or matrices into factors;
- `as.matrix()` turns a vector, or data frame into a matrix;
- `as.vector()` turns matrices into vectors;
- `as.data.frame()` turns vectors and matrices into data frames;
- `as.list()` turns vectors and matrices into lists.

In order to integrate the main database management systems in R one can use the following self-explaining interfaces:

- `RODBC`
- `RMySQL`
- `ROracle`
- `RJDBC`

These interfaces provide access to various kind of databases.

3 Graphical procedures

Graphical facilities are an important and extremely versatile component of the R environment. It is possible to use the facilities to display a wide variety of statistical graphs and also to build entirely new types of graph.
Plotting commands are divided into three basic groups:

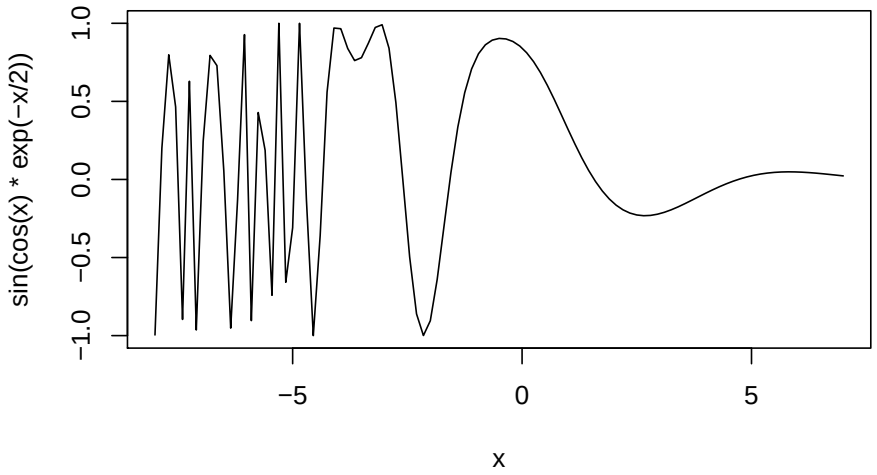
- high-level plotting functions that create a new plot on the graphics device, possibly with axes, labels, titles and so on; in particular, `plot()`, `pie()`, `hist()`, `pairs()`, `boxplot()`, `barplot()`, etc.;
- low-level plotting functions that add more information to an existing plot, such as extra points, lines and labels; in particular, `points()`, `lines()`, `abline()`, `title()`, etc.;
- interactive graphics functions that allow you interactively add information to, or extract information from, an existing plot, using a pointing device such as a mouse.

One of the most frequently used plotting functions in R is the `plot()` function. This is a generic function, since the type of plot produced is dependent on the type or class of the first argument. For example,

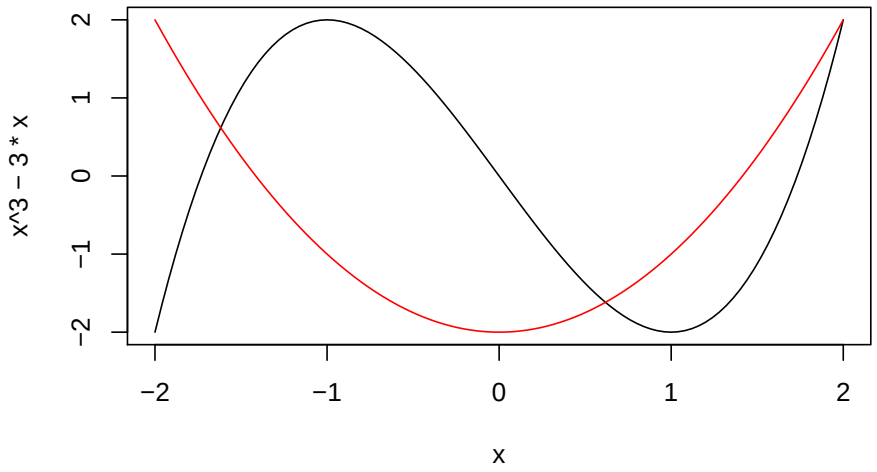
- `plot(x,y)`, if `x` and `y` are vectors, produces a scatterplot of `y` against `x`;
- `plot(x)`, if `x` is a time series, produces a time-series plot; if `x` is a numeric vector, it produces a plot of the values in the vector against their index in the vector.

The function `curve()` draws a curve corresponding to a function over the interval `[from,to]`.

```
curve(sin(cos(x)*exp(-x/2)),from=-8,to=7)
```



```r
# high-level plotting command
curve(x^3-3*x,-2,2)
# low-level plotting command, since with the option add=T the curve is added
# to an already existing plot
curve(x^2-2,add=T,col="red")
```



The function `persp()` draws perspective plots of a surface over the plane, while function `contour()` creates a contour plot, or add contour lines to an existing plot.

4 Writing your own functions

The R language allows the user to create objects of mode `function`. These are true R functions that are stored in a special internal form and they 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. Most of the functions supplied as part of the R system, such as `mean()`, `var()` and so on, are themselves written in R and thus do not differ materially from user written functions.

4.1 Functions

A function is defined by an assignment of the form

```r
name<-function(arg_1,arg_2,...) {expression_1;expression_2;...;return(expression)}
```

With `expression` we indicate an R expression (usually grouped expressions) that uses the argument `arg` (usually more than one) to calculate a value. The value returned for the function is that one given by the command `return(expression)`; otherwise it is given by the result of the last expression. Whenever only one expression is present, the brackets are not necessary.

A call to the function then usually takes the form `name(expr_1,expr_2,...)` and may occur anywhere a function call is legitimate.
Note that any ordinary assignments done within the function are local and temporary and are lost after exit from the function.
`NULL` represents the null object and it is a reserved word. It is often returned by expressions and functions whose value is undefined.
The function giving the BMI (Body Mass Index) is defined below and it is used to obtain the BMI value associated to specific measurements for weight in kilograms and height in metres.

```
# Body Mass Index, weight in kilograms and height in metres
bmi<-function(weight,height)
  {
        x<- weight/height^2
        return(x)
  }

bmi(75,1.7)

[1] 25.95156

x<-c(60,65,75,80,90)
y<-c(1.5,1.6,1.7,1.8,1.9)
bmi(x,y)

[1] 26.66667 25.39062 25.95156 24.69136 24.93075

bmi(1.7,y)

[1] 0.7555556 0.6640625 0.5882353 0.5246914 0.4709141
```

4.2 Grouped expressions

R is an expression language in the sense that its only command type is a function or expression which returns a result. Even an assignment is an expression whose result is the value assigned, and it may be used wherever any expression may be used. In particular, multiple assignments are possible.
Commands may be grouped together in curly brackets, `{expr_1;...;expr_m}`, in which case the value of the group is the result of the last expression in the group evaluated. Since such a group is also an expression it may, for example, be itself included in parentheses and used as a part of an even larger expression, and so on.

4.3 Control statements: conditional execution

The language has available a conditional construction, called `if statement`, of the following forms.

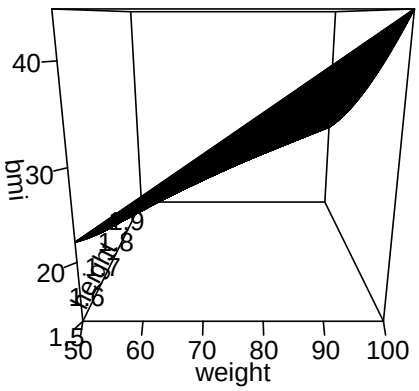
```
if (expr_1) expr_2
if (expr_1) expr_2 else expr_3
```

Here `expr_1` must evaluate to a single logical value and the result of the entire expression is then evident.
The operators `&&` and `||` are often used as part of the condition in an `if statement`. As emphasized before, whereas `&` and `|` apply element-wise to vectors, `&&` and `||` apply to vectors of length one, and only evaluate their second argument if necessary.
We compute the BMI, considering the possibility that the height could be in centimetres.

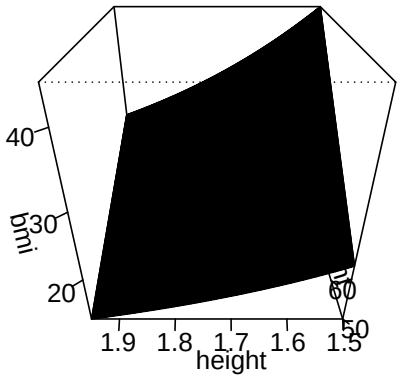
```
# Body Mass Index where height could be in centimetres
bmi1<-function(weight,height,cm=F)
  {
        if(cm==T) height<-height/100
        weight/height^2
  }

bmi1(75,1.7)

[1] 25.95156

bmi1(75,170,cm=T)

[1] 25.95156
```

Using functions `perps()` and `contour()` we obtain the following graphical description of the BMI.

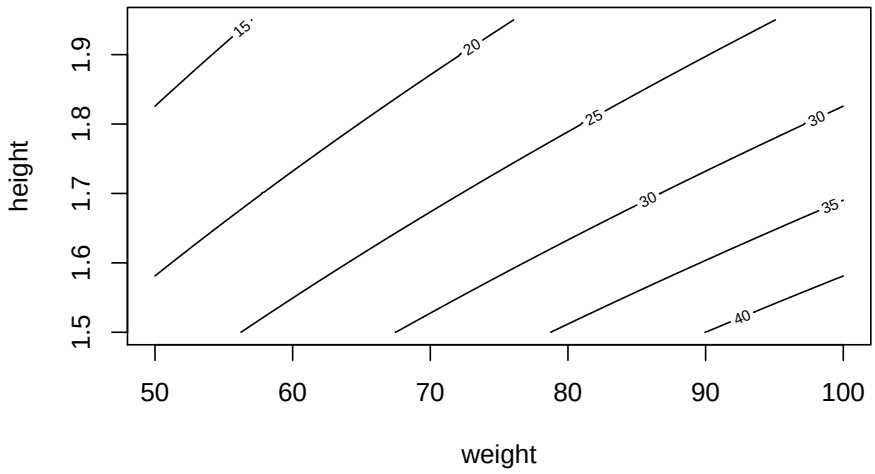
```
x<-seq(50,100,by=0.1)
y<-seq(1.5,1.95, by=0.01)
index<-outer(x,y,bmi)
# 3d graphical representation
persp(x,y,index,xlab="weight",ylab="height",zlab="bmi",ticktype ="detailed")
```



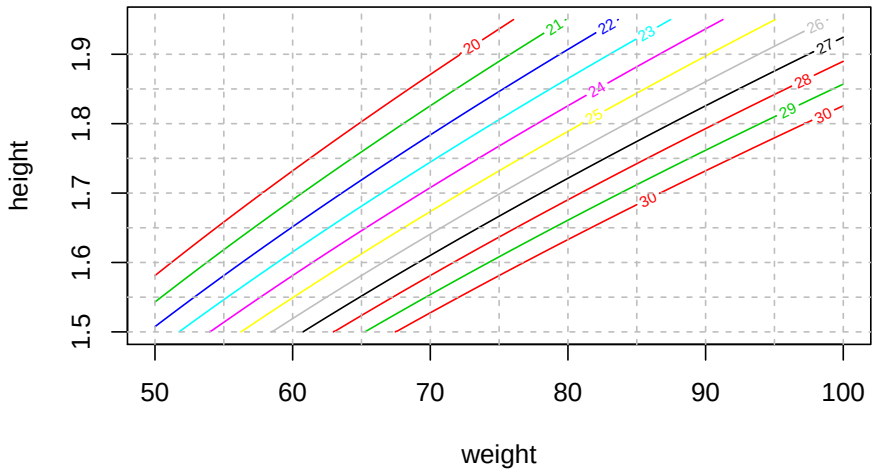

```
persp(x,y,index,xlab="weight",ylab="height",zlab="bmi",theta=-90,phi=30,
      ticktype ="detailed") # rotation
```




```
contour(x,y,index,xlab="weight",ylab="height") # contour plot
```




```
# contour plot with BMI levels from 20 to 30 having different colours
contour(x,y,index,xlab="weight",ylab="height",col=2:11,levels=20:30)
# to add a grid
abline(h=seq(1.5,1.9,by=0.05),lty=2,col="grey")  # low level plot function
abline(v=seq(50,100,by=5),lty=2,col="grey")
```



4.4 Control statements: repetitive execution

The `for loop` construction has the following form.

```
for (name in expr_1) expr_2
```

Here `name` is the loop variable, `expr_1` is a vector expression (often a number sequence such as `1:20`) and `expr_2` is often a group of expressions with its sub-expressions written in terms of the loop variable `name`. Thus `expr_2` is repeatedly evaluated as `name` ranges through the values in the vector result of `expr_1`.
The `for statement` is used in R code much less often than in compiled languages. Code that takes a *whole object* view is likely to be both clearer and faster in R.
The following function computes the first terms of a geometric series with given initial value and ratio.

```
# the first n terms of a geometric series with initial value a0 and ratio r
geom<-function(n,r,a0)
  {
        ser<-numeric(n)
        ser[1]<-a0
        for (i in 2:n)
        ser[i]<-ser[i-1]*r
        return(ser)
  }

geom(10,0.5,1)
```

```
 [1] 1.000000000 0.500000000 0.250000000 0.125000000 0.062500000
 [6] 0.031250000 0.015625000 0.007812500 0.003906250 0.001953125
```

Other looping facilities include the `repeat statement`, having the following simple form.

```
repeat expr
```

Moreover, the `while statement` is defined as follows.

```
while (condition) expr
```

The `break statement` can be used to terminate any loop, possibly abnormally. This is the only way to terminate a `repeat loop`. Finally, the `next statement` can be used to discontinue one particular cycle and to skip to the next one.