R introduction

Release

Maite Ceballos (IFCA), Nicolás Cardiel (UCM)

April 07, 2014

CONTENTS

First steps in R does not pretend to be a comprehensive guide to R package (there are many excellent books and web tutorials) but it aims at providing an introduction to the R statistical package for the (under)graduate students following an introductory Statistics Course.

CHAPTER

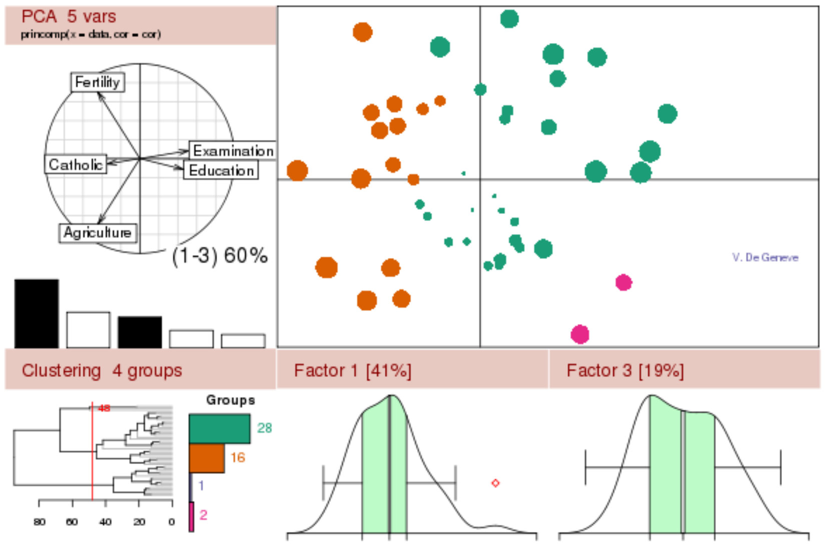
ONE

INTRODUCTION

Why R?

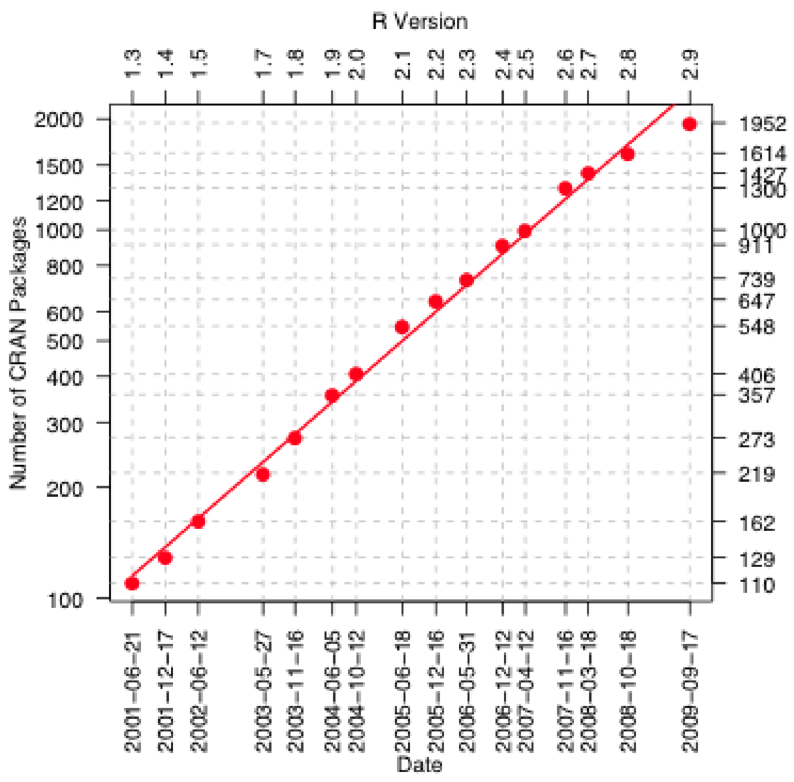
Although there are many tools that can be employed for statistical analysis (SAS, SPSS, Stata, Minitab, MATLAB, Wolfram Mathematica,... among others), we have chosen R because:

- It is an **integrated environment**: it has been developed as a whole entity and not as a collection of tools. It includes:
  - An efficient system for data storage and manipulation
  - A collection of tools to manage arrays
  - Integrated tools for data analysis
  - Screen graphs and portable format graphs generation
  - A simple and effective programming language (with *scripting* capabilities)
- It is **free software**! Available through the project WEB or through CRAN (*Comprehensive R Archive Network*)
- Available for **different platforms** (source code and pre-compiled binaries): *UNIX*, *MacOS*, *Windows*
- ...
- Many scientists are using it!



(*image from* R project web page)

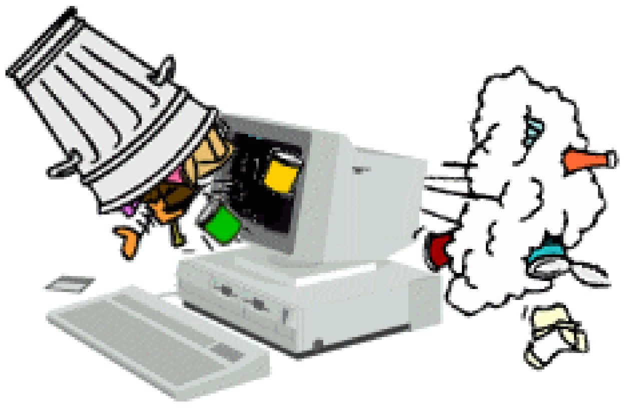
R is continuously (and exponentially) growing with the addition of contributed packages.



(*from* R Journal)

Although... No statistical package can work miracles!

(**GIGO**: Garbage In, Garbage Out)



(*image from* http://www.lovemytool.com)

CHAPTER

TWO

MAIN FEATURES OF R

Note: Before starting to work with R, it is advised to create a new dedicated directory where all the work should be included. In fact, if several projects are to be developed at the same time, every project should have its own directory.

For linux:

```
[user@pc]$ mkdir work
[user@pc]$ cd work
[user@pc work]$ R
```

Warning: R is case-sensitive, thus in this example:

```
> a = 1
> A = 2
```

variables **a** and **A** are different variables:

```
> a == A                                        # is variable 'A' equal to variable 'a' ?
[1] FALSE
```

2.1 Starting R

For linux:

```
[user@pc]$ R                                # invoke R

R version 3.0.1 (2013-05-16) -- "Good Sport"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-redhat-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
```

```
>                                     #  R command line prompt

[user@pc]$ R --silent                 #  Suppress welcome message
[user@pc]$ R --help                   #  Show R options

Usage: R [options] [< infile] [> outfile]
  or: R CMD command [arguments]

Start R, a system for statistical computation and graphics, with the
specified options, or invoke an R tool via the 'R CMD' interface.

Options:
  -h, --help            Print short help message and exit
  --version             Print version info and exit
  --encoding=ENC        Specify encoding to be used for stdin
...
...
```

2.2 Quitting R

```
> quit()
Save workspace image? [y/n/c]:       # possibility of saving info for next session
> quit(save="no")                    # finish R without any question
> Ctrl-D                             # key combination equivalent to quit()
```

Using parenthesis in **quit()** informs R that the command refers to a function and not to a variable.

2.3 Help in R

```
> help.start()                        # general help displayed in a web browser
> help("pp")                          # help on function "pp"
> ?pp                                 # help on function "pp"
> help.search("pp")                   # search for instances of the string "pp"
> ??pp                                # search for instances of the string "pp"
> apropos("pp", mode="function")      # list available functions with "pp" in their names
> example(topic)                      # run the R code from the *Examples* part of R's
                                      #   online help on topic; try for example example(plot)
```

`Tab` key can be used to complete the commands:

```
> Sys.<Tab><Tab>                      # pressing <Tab> twice (after typing 'Sys.') to show
                                      # available Sys options
Sys.chmod         Sys.glob          Sys.setFileTime   Sys.umask
Sys.Date          Sys.info          Sys.setlocale     Sys.unsetenv
Sys.getenv        Sys.localeconv    Sys.sleep         Sys.which
Sys.getlocale     Sys.readlink      Sys.time
Sys.getpid        Sys.setenv        Sys.timezone

> Sys.Date()
[1] "2030-01-01"
```

2.4 Other useful commands

```
> R.version.string
[1] "R version 3.0.1 (2013-05-16)"
```

```
> capabilities()
    jpeg      png     tiff    tcltk      X11     aqua http/ftp  sockets
    TRUE     TRUE     TRUE     TRUE     TRUE    FALSE     TRUE     TRUE
  libxml     fifo   cledit    iconv      NLS  profmem    cairo
    TRUE     TRUE     TRUE     TRUE     TRUE    FALSE     TRUE
> citation()

To cite R in publications use:

  R Core Team (2013). R: A language and environment for statistical
  computing. R Foundation for Statistical Computing, Vienna, Austria.
  URL http://www.R-project.org/.
...
...

> R.home()                            # return the R 'home' directory
[1] "/usr/lib64/R"
> getwd()                             # return the working directory
[1] "/home/user/R"
> setwd("/home/user/newRdir")         # set new working directory
> dir()                               # show content of current directory
...                                   # (different from 'ls()' command
...                                   # which lists objects in current workspace)

> history(n)                          # display the last 'n' commands (default = 25)
...
...                                   # (press "q" to EXIT)

> source("filename.R")                # execute commands in the filename.R script

> sink("register.txt")                # divert R output to an external file

> sink()                              # stop sink-ing (results return to console)
```

CHAPTER

THREE

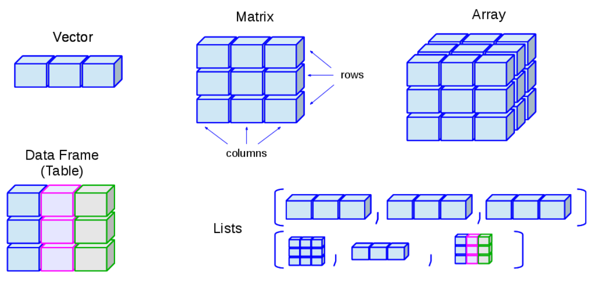
DATA STRUCTURE

R is an object-oriented language: an **object** in R is anything (constants, data structures, functions, graphs) that can be assigned to a variable:

- Data Objects: used to store real or complex numerical values, logical values or characters. These objects are always vectors: there are no scalars in R.
- Language Objects: functions, expressions

3.1 Data structure types

- Vectors: one-dimensional arrays used to store collection data of the same mode
  - Numeric Vectors (mode: *numeric*)
  - Complex Vectors (mode: *complex*)
  - Logical Vectors (model: *logical*)
  - Character Vector or text strings (mode: character)
- Matrices: two-dimensional arrays to store collections of data of the same mode. They are accessed by two integer indices.
- Arrays: similar to matrices but they can be multi-dimensional (more than two dimensions)
- Factors: vectors of categorical variables designed to group the components of another vector with the same size
- **Lists**: ordered collection of objects, where the elements can be of different types
- Data Frames: generalization of matrices where different columns can store different mode data.
- Functions: objects created by the user and reused to make specific operations.



3.1.1 Vectors

Numeric Vectors

There are several ways to assign values to a variable:

```
> a <- 1.7               # assign a value to a vector with only one element (~ scalar)
> 1.7 -> a               # assign a value to a vector with only one element (~ scalar)
> a = 1.7                # assign a value to a vector with only one element (~ scalar)
> assign("a", 1.7)       # assign a value to a vector with only one element (~ scalar)
```

To show the values:

```
> a                      # show the value in the screen (not valid in scripts)
[1] 1.7
> print(a)               # show the value in the screen (valid in scripts)
[1] 1.7
```

To generate a vector with several numeric values:

```
> a <- c(10, 11, 15, 19) # assign four values to a vector using the concatenate command c()
> a                      # show the value in the screen
[1] 10 11 15 19
```

The operations are always done over all the elements of the numeric array:

```
> a*a                    # evaluate the square value of every element in the vector
[1] 100 121 225 361
> 1/a                    # evaluate the inverse value of every element in the vector
[1] 0.10000000 0.09090909 0.06666667 0.05263158
> b <- a-1               # subtract 1 from every element and assign the result to b
> b
[1]  9 10 14 18
```

To generate a *sequence*:

```
> 2:10                            # generate a sequence from n1=2 to n2=10 using n1:n2
[1]  2  3  4  5  6  7  8  9 10
> 5:1                             # generate an inverse sequence if n2 < n1
[1] 5 4 3 2 1

> seq(from=n1, to=n2, by=n3)      # generate sequence from n1 to n2 using n3 step
                                  #  (parameters names can be avoided if order is kept)
> seq(from=1, to=10, by=3)
[1]  1  4  7 10
> seq(1, 10, 3)
[1]  1  4  7 10

> seq(length=10, from=1, by=3)    # generate a fixed length sequence
[1]  1  4  7 10 13 16 19 22 25 28

> help(seq)                       # for help about this command
...
```

To generate *repetitions*:

```
> a <- 1:3; b <- rep(a, times=3); c <- rep(a, each=3)         # command rep()
```

In the previous example we have run three commands in the same line. They have been separated by a ';'.

The content of the three variables is now:

```
> a
[1] 1 2 3
> b
```

```
[1] 1 2 3 1 2 3 1 2 3
> c
[1] 1 1 1 2 2 2 3 3 3
```

The recycling rule: vectors of different sizes can be combined, as far as the length of the longer vector is a multiple of the shorter vector's length (otherwise a warning is issued, although the operation is carried out):

```
> a+c                                            # proper dimensions
[1] 2 3 4 3 4 5 4 5 6                            # (operation equivalent to b+c)

> d <- c(10,100)
> b+d                                            # incorrect dimensions
[1]  11 102  13 101  12 103  11 102  13
Warning message:
In b + d : longer object length is not a multiple of shorter object length
```

If we need to know which are the objects that are currently defined, we can *list* them:

```
> ls()
[1] "a" "b" "c" "d"
```

Undesired objects can be deleted using `rm()` function:

```
> rm(a,c)                                        # remove objects 'a' and 'b'
> ls()                                           # list current objects
[1] "b" "d"
```

In order to remove everything in the working environment:

```
> rm(list=ls())                                  # Use this with caution
> ls()                                           # (you'll receive no warning!)
character(0)
```

Logical Vectors

```
> a <- seq(1:10)                                 # generate a sequence
> a
[1]  1  2  3  4  5  6  7  8  9 10                # show values in screen
> b <- (a>5)                                     # assign values from an inequality
> b                                              # show values in screen
[1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> a[b]                                           # show values that fulfil the condition
[1]  6  7  8  9 10
> a[a>5]                                         # the same, but avoiding intermediate variable
[1]  6  7  8  9 10
```

Character Vectors

```
> a <- "This is an example"                      # generate a character vector
> a                                              # show vector content
[1] "This is an example"
```

We can concatenate vectors after converting them into character vectors:

```
> x <- 1.5
> y <- -2.7
> paste("Point is (",x,",",y,")", sep="")        # concatenate x, y and a string using 'paste'
[1] "Point is (1.5,-2.7)"
```

3.1.2 Matrices

A matrix is a **bi-dimensional** collection of data:

```
> a <- matrix(1:12, nrow=3, ncol=4)           # define a matrix with 3 rows and 4 columns
> a
   [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> dim(a)                                      # return matrix dimensions (rows,columns)
[1] 3 4
```

The elements of vectors and matrices are **recycled** when it is required by the involved dimensions:

```
> a <- matrix(1:8, nrow=4, ncol=4)            # create a matrix with 4 rows and 4 columns
> a
     [,1] [,2] [,3] [,4]
[1,]    1    5    1    5
[2,]    2    6    2    6
[3,]    3    7    3    7
[4,]    4    8    4    8
```

3.1.3 Arrays

They are similar to the matrices although they can have 2 o more dimensions.

```
> z <- array(1:24, dim=c(2,3,4))
> z
, , 1

      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

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

, , 3

      [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 4

      [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
```

3.1.4 Factors

Factors are vectors that contain categorical information useful to group the values of other vectors of the same size. Let's see an example:

```
> bv <- c(0.92,0.97,0.87,0.91,0.92,1.04,0.91,0.94,0.96,
+         0.90,0.96,0.86,0.85)                        # (B-V) colours from 13 galaxies
```

If additional information is available (for instance, the morphological type of the galaxies) we can create a **factor** containing the galaxy types:

```
> morfo <- c("Sab","E","Sab","S0","E","E","S0","S0","E",
+            "Sab","E","Sab","S0")                    # morphological info (same size)
> length(morfo)                                       # ensure vector is the same size
[1] 13
> fmorfo <- factor(morfo)                             # create factor with 'factor()'
> fmorfo
[1] Sab E   Sab S0  E   E   S0  S0  E   Sab E   Sab S0        # show factor content
Levels: E S0 Sab                                      # factor different values (levels)
> levels(fmorfo)                                      # show factor levels
[1] "E"   "S0"  "Sab"
```

We could use this additional information to perform an statistical analysis segregating the data according to these types. This will be covered lately in the *Functions* section.

3.1.5 Lists

Lists are ordered collections of objects, where the elements can be of a different type (a list can be a combination of matrices, vectors, other lists, etc.) They are created using the `list()` function:

```
> gal <- list(name="NGC3379", morf="E", T.RC3=-5, colours=c(0.53,0.96))
> gal
$name
[1] "NGC3379"

$morf
[1] "E"

$T.RC3
[1] -5

$colours
[1] 0.53 0.96

> gal$<Tab>                  # pressing Tab key after '$', the elements of 'gal' are shown
gal$name     gal$morf     gal$T.RC3    gal$colours

> length(gal)                # check how many elements 'gal' has
[1] 4

> names(gal)                 # return element names
[1] "name"     "morf"     "T.RC3"    "colours"
```

New elements can be added in a simple way, just defining them:

```
> gal$radio <- TRUE                          # add a boolean element
> gal$redshift <- 0.002922                   # add a numeric element

> names(gal)                                 # return element names
[1] "name"      "morf"      "T.RC3"     "colours"  "radio"     "redshift"
```

Lists can be concatenated to generate bigger lists. If we have `list1`, `list2`, `list3`, we can create a unique list which is the result of the union of these three lists:

```
> list123 <- c(list1, list2, list3)
```

As the elements in a list can be R objects of a different type:

- Lists are extremely versatile since they can store every type of information (good)
- Lists can be converted in objects with a rather complex structure (bad). A list can contain several elements which are vectors of different length, which is similar to having a table where the columns have a different

number of rows.

The ideal situation is to take advantage of the list versatility but preventing them from growing with a very complex structure. This is why R has defined a new type of data which fulfils both requirements: a Data Frame.

3.1.6 Data Frames (Tables)

A *Data Frame* is an special type of list very useful for the statistical work. There are some restrictions to guarantee that they can be used for this statistical purpose.

Among other restrictions, a *Data Frame* must verify that:

- List components must be vectors (numeric, character or logical vectors), factors, numeric matrices or other data frames.
- Vectors, which are the variables in the data frame, must be of the same length.

Warning: In a data frame, character vectors are automatically converted into factors, and the number of levels can be determined as the number of different values in such a vector. This default behaviour can be modified with the `options(stringsAsFactors = FALSE)` command.

Basically, in a *Data Frame* all the information is displayed as a **table** where the columns have the same number of rows and can contain different type objects (numbers, characters, ...).

Data Frames can be created using the `data.frame()` function. Let's see how to define a *data frame* with two elements, a numeric vector and a character vector (note that both must be same length vectors):

```
> options(stringsAsFactors = FALSE)
> df <- data.frame(numbers=c(10,20,30,40),text=c("a","b","c","a"))
> df
  numbers text
1      10    a
2      20    b
3      30    c
4      40    a
> df$text                                    # character vector not converted to a factor
[1] "a" "b" "c" "a"


> options(stringsAsFactors = TRUE)           # default
> df <- data.frame(numbers=c(10,20,30,40),text=c("a","b","c","a"))
> df$text
[1] a b c a                                  # character vector of length = 4
Levels: a b c                                #  converted to a three levels factor!!
> df$numbers
[1] 10 20 30 40                              # numeric vector of length = 4

> mode(df)                                   # storage mode of the object
[1] "list"
> typeof(df)                                 # (internal) storage mode of the object
[1] "list"
> class(df)                                  # object class
[1] "data.frame"
```

However the most common way of defining a *data frame* is reading the data stored in a file. We will see later how to do it using `read.table()` function.

Factors and Tables

It is frequently useful (for instance, for table creation) to be able to generate factors from a numeric continuum variable. To do so, we can use the `cut` command. Its parameter `breaks` defines how the data are divided. **If** `breaks` **is a number**, this is used as the number of (same length) intervals:

```
> bv <- c(0.92,0.97,0.87,0.91,0.92,1.04,0.91,0.94,0.96,
+         0.90,0.96,0.86,0.85)                  # (B-V) colors from 13 galaxies
> fbv <- cut(bv,breaks=3)                       # divide 'bv' in 3 equal-length intervals
> fbv                                           # show in which interval every galaxy is
[1] (0.913,0.977] (0.913,0.977] (0.85,0.913]  (0.85,0.913]  (0.913,0.977]
[6] (0.977,1.04]  (0.85,0.913]  (0.913,0.977] (0.913,0.977] (0.85,0.913]
[11] (0.913,0.977] (0.85,0.913]  (0.85,0.913]
Levels: (0.85,0.913] (0.913,0.977] (0.977,1.04]        # the 3 intervals
> table(fbv)                                    # generate a table with the 3 intervals
fbv
  (0.85,0.913] (0.913,0.977]  (0.977,1.04]
           6             6             1
```

If `breaks` **is a vector**, its values are used as the limits of the intervals:

```
> ffbv <- cut(bv,breaks=c(0.80,0.90,1.00,1.10))
> table(ffbv)
ffbv
  (0.8,0.9]    (0.9,1]    (1,1.1]
       4          8          1
```

If we want just an approximate number of intervals, but with equally spaced *round* values, we can use the `pretty()` function (that not always returns the specified number of intervals!):

```
> fffbv <- cut(bv,pretty(bv,3))                 # ask for 3 'pretty' intervals
> table(fffbv)                                  # return 4 intervals
fffbv
  (0.85,0.9] (0.9,0.95]   (0.95,1]   (1,1.05]
        3          5          3          1
```

We can also use a quantile division:

```
> ffffbv <- cut(bv,quantile(bv,(0:4)/4))        # ask for the 4 quantiles
> table(ffffbv)
ffffbv
  (0.85,0.9]   (0.9,0.92] (0.92,0.96] (0.96,1.04]
         3            4           3           2
```

Warning: The last two groupings exclude the value 0.85 which is one of our data values.

Factors can be used to build multi-dimensional tables. Let's see how. First of all, we will define the data (that in a real case would be read from a data file):

```
> heights <- c(1.64,1.76,1.79,1.65,1.68,1.65,1.86,1.82,1.73,
+              1.75,1.59,1.87,1.73,1.57,1.63,1.71,1.68,1.73,1.53,1.82)
> weights <- c(64,77,82,62,71,72,85,68,72,75,81,88,72,
+              71,74,69,81,67,65,73)
> ages <- c(12,34,23,53,23,12,53,38,83,28,28,58,38,
+           63,72,44,33,27,32,38)
```

For each one of these variables we can generate factors:

```
> fheights <- cut(heights,c(1.50,1.60,1.70,1.80,1.90))         # factor for 'heights'
> fweights <- cut(weights,c(60,70,80,90))                      # factor for 'weights'
> fages    <- cut(ages,seq(10,90,10))                          # factor for 'ages'
```

Table generation is now straightforward using these factors. We can, for instance, generate bi-dimensional tables:

```
> ta <- table(fheights, fweights)                       # table for 'heights' vs. 'weights'
> ta
           fweights
fheights    (60,70] (70,80] (80,90]
  (1.5,1.6]       1       1       1
  (1.6,1.7]       2       3       1
  (1.7,1.8]       2       4       1
  (1.8,1.9]       1       1       2
```

Marginal frequencies can also be included:

```
> addmargins(ta)
           fweights
fheights    (60,70] (70,80] (80,90] Sum
  (1.5,1.6]       1       1       1   3
  (1.6,1.7]       2       3       1   6
  (1.7,1.8]       2       4       1   7
  (1.8,1.9]       1       1       2   4
  Sum             6       9       5  20
```

Or we can work with the relative frequencies;

```
> tta <- prop.table(ta)
> addmargins(tta)
         fweights
fheights    (60,70] (70,80] (80,90]  Sum
  (1.5,1.6]    0.05    0.05    0.05 0.15
  (1.6,1.7]    0.10    0.15    0.05 0.30
  (1.7,1.8]    0.10    0.20    0.05 0.35
  (1.8,1.9]    0.05    0.05    0.10 0.20
  Sum          0.30    0.45    0.25 1.00
```

We can also generate tridimensional tables. Following the previous example, we can examine the same bi-dimensional table for each age interval:

```
> table(fheights, fweights, fages)
, , fages = (10,20]                                  # first age interval

         fweights
fheights    (60,70] (70,80] (80,90]
  (1.5,1.6]       0       0       0
  (1.6,1.7]       1       1       0
  (1.7,1.8]       0       0       0
  (1.8,1.9]       0       0       0

, , fages = (20,30]                                  # second age interval

         fweights
fheights    (60,70] (70,80] (80,90]
  (1.5,1.6]       0       0       1
  (1.6,1.7]       0       1       0
  (1.7,1.8]       1       1       1
  (1.8,1.9]       0       0       0

........

, , fages = (70,80]                                  # next-to-the-last age interval

         fweights
fheights    (60,70] (70,80] (80,90]
  (1.5,1.6]       0       0       0
  (1.6,1.7]       0       1       0
  (1.7,1.8]       0       0       0
  (1.8,1.9]       0       0       0

, , fages = (80,90]                                  # last age interval

         fweights
fheights    (60,70] (70,80] (80,90]
  (1.5,1.6]       0       0       0
  (1.6,1.7]       0       0       0
  (1.7,1.8]       0       1       0
  (1.8,1.9]       0       0       0

> sum(table(fheights, fweights, fages))               # check total number of entries
[1] 20
```

Matrices and Tables

We can easily generate 2D tables from matrices:

```
> mtab <- matrix(c(30,12,47,58,25,32), ncol=2, byrow=TRUE)    # create a matrix filled by rows
> colnames(mtab) <- c("ellipticals","spirals")                # set matrix column names
> rownames(mtab) <- c("sample1","sample2","new sample")       # set matrix row names
> mtab
           ellipticals spirals
sample1             30      12
sample2             47      58
new sample          25      32
```

However, `mtab` is not a true R table. To transform it into a true table we can use:

```
> rtab <- as.table(mtab)

> mode(mtab);mode(rtab)                               # indistinguishable in 'mode'
[1] "numeric"
[1] "numeric"

> typeof(mtab);typeof(rtab)                           # indistinguishable in 'typeof'
[1] "double"
[1] "double"

> class(mtab);class(rtab)                             # but different in 'class' !
[1] "matrix"
[1] "table"
```

In addition to the functions to calculate *marginal distributions* (`margin.table`), *frequencies* (`prop.table`), etc., the command `summary` returns the χ^2 test for the independence of the factors:

```
> summary(rtab)
Number of cases in table: 204
Number of factors: 2
Test for independence of all factors:
        Chisq = 9.726, df = 2, p-value = 0.007726
```

The same command returns a different result when it is applied to a matrix type object:

```
> summary(mtab)
        V1              V2
  Min.   :25.0   Min.   :12
  1st Qu.:27.5   1st Qu.:22
  Median :30.0   Median :32
  Mean   :34.0   Mean   :34
  3rd Qu.:38.5   3rd Qu.:45
  Max.   :47.0   Max.   :58
```

3.1.7 Functions

These are objects that can be created by the user and then re-used to make specific operations.

For example, we can define a **function** to calculate the standard deviation:

```
> stddev <- function(x) {                      # user-defined function 'stddev'
+   res = sqrt(sum((x-mean(x))^2) / (length(x)-1))
+   return(res)
+ }
```

Functions can be defined inside other functions (nested) and can also be passed as arguments to other functions. The value returned by a function is the result of the last expression evaluated in the body of the function or the value grabbed by the `return` command.

R functions arguments can have *default values* or can be *missing*. Arguments can be matched by name or position:

```
> mynumbers <- c(1, 2, 3, 4, 5)
> stddev(mynumbers)                             # equivalent calls to 'stddev'
[1] 1.581139
> stddev(x = mynumbers)
[1] 1.581139

> sd(x=mynumbers)                               # R function using 'missing argument' with
[1] 1.581139                                    #       default value (FALSE)
> sd(x=mynumbers, na.rm=TRUE)                   # Specify all arguments by name
[1] 1.581139
> sd(mynumbers, na.rm=TRUE)                     # Mixing positional and by name matching
[1] 1.581139
> sd(na.rm=TRUE, x=mynumbers)                   # legal but not recommended (keep order)
[1] 1.581139
```

Looping Functions

There are special R functions that can be used to repeat instructions in the command line and facilitate the programming process:

- **lapply**: evaluate a function for each element of a list
- **sapply**: evaluate a function for each element of a list *simplifying* the result
- **apply**: Apply a function over the margins of an array (usually to apply a function to the rows/columns in a matrix)
- **tapply**: Apply a function over subsets of a vector (for example defined with a factor)
- **mapply**: Multivariate version of lapply

Let's see how to apply these functions to the previous example with the galaxy colours:

```
> bv.vec <- c(0.92,0.97,0.87, 0.91,0.92,1.04,0.91,0.94,0.96,
+             0.90,0.96,0.86,0.85)                      # (B-V) colours from 13 galaxies
> morfo <- c("Sab","E","Sab","S0","E",  "E","S0","S0","E",    # ordered morph. information
+            "Sab","E","Sab","S0")                      #    for the galaxies
```

lapply

```
> bv.list <- list(colsSab=c(0.92,0.87,0.90,0.86),
+                 colsE=c(0.97,0.92,1.04,0.96,0.96),
+                 colsSO=c(0.91,0.91,0.94,0.85))

> lapply(bv.list, mean)                         # calculate mean for each galaxy type
$colsSab                                        #    (returns a list)
[1] 0.8875

$colsE
[1] 0.97

$colsSO
[1] 0.9025
```

sapply

```
> sapply(bv.list, mean)                             # simplified version of 'lapply'
colsSab   colsE  colsSO                             #    (returns a vector)
0.8875  0.9700  0.9025
```

tapply

```
> fmorfo <- factor(morfo)                           # create factor
> tapply(bv,fmorfo,mean)                            # apply mean function to the galaxy colours
     E     S0    Sab                                #    segregating by morphological type
0.9700 0.9025 0.8875
```

apply

```
> a <- matrix(1:12, nrow=3, ncol=4)                 # define a matrix with 3 rows and 4 columns
> a
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> apply(a,1,mean)                                   # calculate rows ("1") mean == rowMeans
[1] 5.5 6.5 7.5
> rowMeans(a)
[1] 5.5 6.5 7.5

> apply(a,1,sum)                                    # calculate rows ("1") sum == rowSums
[1] 22 26 30
> rowSums(a)
[1] 22 26 30

> apply(a,2,mean)                                   # calculate columns ("2") mean == colMeans
[1]  2  5  8 11
> apply(a,2,sum)                                    # calculate columns ("2") sum == colSums
[1]  6 15 24 33
```

3.2 Special Values

It is useful to define some values as * Not Available* (*NA*):

```
> a <- c(0:2, NA, NA, 5:7)                          # define vector with NA values
> a                                                 # show values in screen
[1]  0  1  2 NA NA  5  6  7
```

We can carry out mathematical operations:

```
> a*a                                               # calculate the square of 'a'
[1]  0  1  4 NA NA 25 36 49
```

We can check whether there is any undefined value:

```
> unavail <- is.na(a)                               # use of is.na() function
> unavail
[1] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
```

Sometimes calculations end up in values with no mathematical sense:

```
> a <- log(-1)
> a
[1] NaN                                             # Result is Not-a-Number (NaN)
> a <- 1/0; b <- 0/0; c <- log(0); d <- c(a,b,c)
> d
```

```
[1]  Inf  NaN -Inf                            # Infinities and Not-a-Number
> 1/Inf                                       # Possible to operate with Infinite
[1] 0                                         #   (if it makes sense!)
```

To check whether we have *Infinite* values or *Not-a-Number* values:

```
> is.infinite(d)                              # is there any Infinite value?
[1]  TRUE FALSE  TRUE
> is.nan(d)                                   # is there any Not-a-Number value?
[1] FALSE  TRUE FALSE
```

Main R functions (`mean`, `var`, `sum`, `min`, `max`,...) accept an argument called `na.rm` that can be set as `TRUE` or `FALSE` to remove (or not) the unavailable data.

```
> a <- c(0:2, NA, NA, 5:7)                    # define vector 'a' with Not-Available data
> a
[1]  0  1  2 NA NA  5  6  7
> mean(a)                                     # since there are Not-Available data
[1] NA

> mean(a, na.rm=TRUE)                         # calculate mean, ignoring Not-Available values
[1] 3.5
```

3.3 Subsetting

Several R operators can be used to extract subsets (slices) from R objects:

- **[** can be used to extract **one or more elements** of an R object. It always returns an object of the same class
- **[[** can be used to extract a **single** element from a data frame or a list. The class of the extracted element can be different from the original object.
- **$** can be used to extract **named** elements from a data frame or a list.

For *Numeric Vectors*:

```
> a <- 1:15                               # generate a sequence
> a <- a*a                                # calculate the square of 'a'
> a                                       # show in screen
[1]   1   4   9  16  25  36  49  64  81 100 121 144 169 196 225
> a[3]                                    # access to the third value in the vector
[1] 9                                     #   (numeric index)
> a[3:5]                                  # access to a continuum slice of values
[1]  9 16 25                              #   (numeric index)
> a[c(1,3,10)]                            # access to a given sequence of values
[1]   1   9 100                           #   (numeric index)
> a[-1]                                   # negative index remove values from vector
[1]   4   9  16  25  36  49  64  81 100 121 144 169 196 225
> a[c(-1,-3,-5,-7)]                       # remove several values (it is not possible
[1]   4  16  36  64  81 100 121 144 169 196 225   to mix positive and negative indices!)
> a[a>100]                                # access to a sequence based on a condition
[1] 121 144 169 196 225                   #   (logical index)
```

For *Character Vectors*:

```
 > a <- c("A", "B", "C", "C", "D", "E")
 > a[1]                                   # first element of "a" (also a character vector)
 [1] "A"                                  #   (numeric index)
 > a[1:4]                                 # sequence of the first 4 elements
 [1] "A" "B" "C" "C"
 > a[a>"C"]                               # select elements "greater" than letter "C"
[1] "D" "E"                               #   (logical index)
 > gtC <- a > "C"                         # the same but using an intermediate logical vector
 > gtC
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE
 > a[gtC]
 [1] "D" "E"
```

For *Matrices*, elements are accessed through two integer indices:

Note: The agreement to establish the indices order `a[i,j]` is the same than the one used in Math for the matrix coefficients a $_{ij}$

```
> a <- matrix(1:12, nrow=3, ncol=4)    # define a matrix with 3 rows and 4 columns
> a
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12


> a[2,3]                               # return the value in the 2nd row and 3th column
[1] 8
> a[[2,3]]                             # return the value in the 2nd row and 3th column
[1] 8
> a[2,]                                # return the values for the second row
[1]  2  5  8 11
> a[,3]                                # return the values for the third column
[1] 7 8 9
```

Note: By default, subsetting a single element or a single row or a single column returns a vector, not a matrix (this can be changed using `drop=FALSE`)

```
> a[2,3, drop=FALSE]                   # so as not to 'drop' the dimension
      [,1]                             #     (returns a 1x1 matrix)
[1,]    8
> a[2, , drop=FALSE]                   # return a 1x4 matrix
      [,1] [,2] [,3] [,4]
[1,]    2    5    8   11
```

The access to the matrix elements can be done with the indices stored in other auxiliary matrices:

```
> ind <- matrix(c(1:3,3:1), nrow=3, ncol=2)    # auxiliary matrix for the indices i,j
> ind
     [,1] [,2]
[1,]    1    3
[2,]    2    2
[3,]    3    1

> a[ind] <- 0                          # set to 0 the matrix values in the indices
> a                                    #  specified in 'ind' (1,3), (2,2), (3,1)
     [,1] [,2] [,3] [,4]
[1,]    1    4    0   10
[2,]    2    0    8   11
[3,]    0    6    9   12
```

For *lists*:

The list components can be accessed using the three operators mentioned above (*[*, *[[* and *$*):

```
> gal <- list(name="NGC3379", morf="E", colours=c(0.53,0.96))

> gal[3]                               # access to the third element of the list
$colours                               #  (get back a list with one element called 'colours'
[1] 0.53 0.96                          #   with the sequence '0.53,0.96')
> gal["colours"]                       # single bracket + name (same as above)
$colours
[1] 0.53 0.96

> gal[[3]]                             # access to the third element of the list
[1] 0.53 0.96                          #   (get back just the sequence)
> gal[["colours"]]                     # double bracket + name (same as above)
[1] 0.53 0.96


> gal$colours                          # element associated with the name 'colours'
[1] 0.53 0.96                          #   (same as double bracket)
> gal$colours[1]                       # first element of the sequence in the third element
[1] 0.53
> gal$colours[2]                       # second element of the sequence in the third element
[1] 0.96
```

To extract **multiple elements** of a list, single bracket is mandatory:

```
> gal <- list(name="NGC3379", morf="E", colours=c(0.53,0.96))

> gal[c(1,2)]                          # return a list with the elements 'name' and 'morf'
$name
[1] "NGC3379"

$morf
[1] "E"
```

For **computed** indices the *[[* and *[* operators can be used. The *$* operator can only be used with *literal* names:

```
> gal <- list(name="NGC3379", morf="E", colours=c(0.53,0.96))

> info <- "morf"                       # variable containing the name of one of the list elements

> gal[["morf"]]
[1] "E"
> gal[[info]]                          # computed index for 'morf' with double bracket
[1] "E"

> gal["morf"]
$morf
[1] "E"
> gal[info]                            # computed index for 'morf' with single bracket
$morf
[1] "E"

> gal$morf
[1] "E"
> gal$info                             # element 'info' unknown
NULL
```

To **recursively** extract an element:

```
> gal <- list(name="NGC3379", morf="E", colours=c(0.53,0.96))

> gal[[c(3,1)]]                        # extract the 1st element of the 3rd element ('0.53')
[1] 0.53
> gal[[3]][[1]]                        # equivalent double subsetting
[1] 0.53

> gal[c(3,1)]                          # not recursive!
$colours
[1] 0.53 0.96
```

```
$name
[1] "NGC3379"
```

Elements can be extracted using **partial matching** with the *[[* and *$* operators:

```
> gal <- list(name="NGC3379", morf="E", colours=c(0.53,0.96))

> gal$na                          # get element by partial matching the name
[1] "NGC3379"
> gal[["na"]]                     # expect exact element name
NULL
> gal[["na", exact=FALSE]]        # partial matching as with '$'
[1] "NGC3379"
```

For *Data Frames (Tables)*, the operators used for slicing are the same than those used for *lists*:

```
> airquality                                # data frame in R library
> airquality[1:7, ]                         # display first 7 rows of data frame
  Ozone Solar.R Wind Temp Month Day         # there are missing values in rows 5 and 6
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
7    23     299  8.6   65     5   7
> class(airquality[1:7, ])
[1] "data.frame"

> airquality[1,1]                           # get element in row=1, col=1
[1] 41
> airquality[[1,1]]                         # get element in row=1, col=1
[1] 41

> airquality[1,]                            # get row=1 (all columns)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
> class(airquality[1,])
[1] "data.frame"
> as.numeric(airquality[1,])                # get row=1 into a numeric vector
[1]  41.0 190.0   7.4  67.0   5.0   1.0


> airquality$Ozone                          # get "Ozone" column into a vector
  [1]  41  36  12  18  NA  28  23  19   8  NA   7  16  11  14  18  14  34   6
 [19]  30  11   1  11   4  32  NA  NA  NA  23  45 115  37  NA  NA  NA  NA  NA
 [37]  NA  29  NA  71  39  NA  NA  23  NA  NA  21  37  20  12  13  NA  NA  NA
 [55]  NA  NA  NA  NA  NA  NA  NA 135  49  32  NA  64  40  77  97  97  85  NA
 [73]  10  27  NA   7  48  35  61  79  63  16  NA  NA  80 108  20  52  82  50
 [91]  64  59  39   9  16  78  35  66 122  89 110  NA  NA  44  28  65  NA  22
[109]  59  23  31  44  21   9  NA  45 168  73  NA  76 118  84  85  96  78  73
[127]  91  47  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

> class(airquality$Ozone)
[1] "integer"
```

For *Character Strings* the access to their elements is done in a different way:

```
> a <- "This is an example of a text string"  # define a character string
> substr(a,5,10)                              # show a string subset
[1] " is an"
```

3.3.1 Removing NA values

We can remove *Not Available* values in a simple way using subsetting:

```
> a <- c(0:2, NA, NA, 5:7)                      # define vector with NA values

> aa <- a[!is.na(a)]                            # the condition uses the negation
> aa                                            #    of is.na() function
[1] 0 1 2 5 6 7                                 # new vector with no NA values
```

To take the subset of multiple vectors avoiding the missing values:

```
> a <- c( 1,  2, 3, NA, 5, NA, 7)
> b <- c("A","B",NA,"D",NA,"E","F")
> valsok <- complete.cases(a,b)                 # return positions in which both vectors have
> valsok                                        #    no-missing values
[1]  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE
> a[valsok]                                     # subsetting 'a' gets good elements in 'a'
[1] 1 2 7
> b[valsok]                                     # subsetting 'b' gets good elements in 'b'
[1] "A" "B" "F"
```

We can also use the function `complete.cases` to remove missing values from data frames:

```
> airquality                                    # data frame in R library
> airquality[1:7, ]                             # display first 7 rows of data frame
  Ozone Solar.R Wind Temp Month Day             # there are missing values in rows 5 and 6
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
7    23     299  8.6   65     5   7

> valsok <- complete.cases(airquality)          # rows in which all the values are ok
> airquality[valsok, ][1:7,]                    # subset original dataframe and show first 7 rows
  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
7    23     299  8.6   65     5   7
8    19      99 13.8   59     5   8
9     8      19 20.1   61     5   9
```

CHAPTER

FOUR

BASIC OPERATIONS

```
> a <- c(7+4,7-4,7*4,7/4)      # elemental arithmetic operations
> a
[1] 11.00  3.00 28.00  1.75

> length(a)                    # return vector length
[1] 4

> c(min(a),max(a))             # calculate minimum and maximum value of the vector
[1]  1.75 28.00

> which.min(a)                 # determine the location (index) of the minimum
[1] 4

> which.max(a)                 # determine the location (index) of the maximum
[1] 3

> sort(a)                      # sort vector values
[1]  1.75  3.00 11.00 28.00

> sum(a)                       # calculate sum of all vector values
[1] 43.75

> cumsum(1:10)                 # calculate cumulative sum
[1]  1  3  6 10 15 21 28 36 45 55

> cumprod(1:5)                 # calculate cumulative product
[1]     1     2     6    24   120   720  5040 40320

> mean(a)                      # calculate the mean value
[1] 10.9375

> median(a)                    # calculate the median value
[1] 7

> var(a)                       # calculate the variance
[1] 146.1823

> sd(a)                        # calculate the standard deviation
[1] 12.09059

> quantile(a, 0.25)            # calculate first quantile (prob=25%)
   25%
2.6875
```

There is a command to get basic statistical information in a simple way:

```
> summary(a)
    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
  1.750   2.688   7.000  10.940  15.250  28.000
```

Some important mathematical functions are `exp()`, `sin()`, `cos()`, `tan()`, `log()`, `log10()`,...

```
> ?Trig                         # show information about trigonometric functions
> ?exp                          # help about 'exp()' function
```

R also includes Special functions of Mathematics: `beta(a,b)`, `gamma(x)`, ...

```
> ?Special                      # help about Special mathematical functions
```

Operations in R can be *vectorized* helping to improve the code readability and efficiency:

```
> a <- seq(10,30,10)
> b <- seq(1:3)
> a + b                         # makes the sum of two vectors
[1] 11 22 33

> a * b                         # vector product
[1] 10 40 90
> a / b                         # vector division
[1] 10 10 10

> a > 5                         # logical operations
[1] TRUE TRUE TRUE
> b == 2
[1] FALSE  TRUE FALSE
```

The vectorization can be also performed over matrices:

```
> m1 <- matrix(1:9, 3, 3)       # 3 x 3 matrix definition
> m1
    [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> m2 <- matrix(11:19, 3, 3)     # 3 x 3 matrix definition
> m2
    [,1] [,2] [,3]
[1,]   11   14   17
[2,]   12   15   18
[3,]   13   16   19

> m1 * m2                       # element-wise matrix multiplication
     [,1] [,2] [,3]
[1,]   11   56  119
[2,]   24   75  144
[3,]   39   96  171

> m1 %*% m2                     # true matrix multiplication
     [,1] [,2] [,3]
[1,]  150  186  222
[2,]  186  231  276
[3,]  222  276  330
```

Examples:

```
Not shown; see web page.
```

CHAPTER

FIVE

CONTROL STRUCTURES

5.1 Types

Böhm and Jacopini's work [1] in 1966, showed that the computer programs can be developed using only three *control structures*:

1. Sequence Structure: the instructions are executed in the sequential order they have been written, unless the contrary is specified. In R, this behaviour is inherent to the interactive execution (through the R interpreter) and it is also the way in which instructions are executed in a script.

2. Selection Structure: different instructions can be executed depending on a condition. In R this is implemented through:

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

3. Repetition Structure: the execution of a group of instructions can be repeated inside a loop. This can be accomplished by:

```
> for (name in expr_1) expr_2
> while (condition) expr
> repeat expr
```

Every algorithm can be resolved using the control structures described above. These structures can be nested so the use of braces "{...}" and proper indentation make the blocks of instructions clearer:

```
for (x in seq(-3,3)) {
  if (x < 0) {
    print("Caso A:")
    y <- sqrt(-x)
    cat("y=",y,"\n")
  } else {
    print("Caso B:")
    y <- sqrt(x)
    cat("y=",y,"\n")
  }
}
```

5.2 Control Flow: break, next, return

These three commands are used to alter the normal execution of the control structures. From R *help*:

break: breaks out of a 'for', 'while' or 'repeat' loop (applies only to the innermost loop).

[1] Böhm C., and Jacopini G,. Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules, Communications of the ACM, Vol 9., No. 5, 1966, 336–371.

next: halts the processing of the current iteration and advances the looping index (applies only to the innermost loop.)

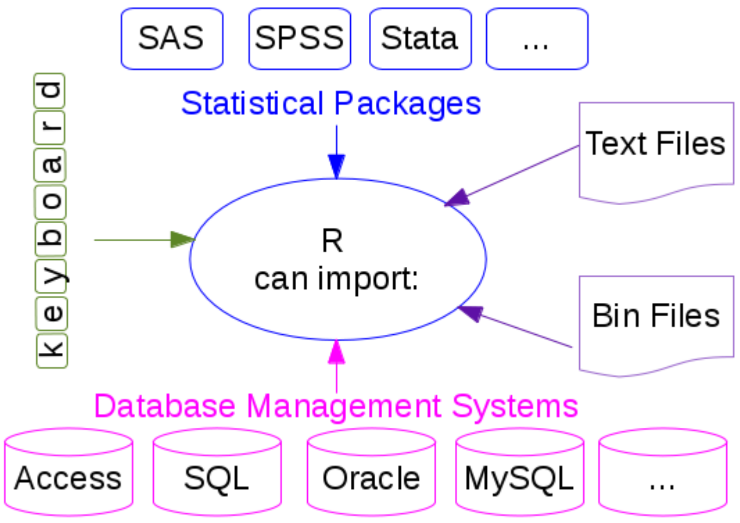
return: returns a value in a function and exits it.

CHAPTER

SIX

DATA READING AND WRITTING

You can import data in R in many different formats!



6.1 ASCII data files

The main functions used in R to import data from ASCII files are `read.table` and `read.csv` to read data in a tabular form, and `readLines` to read lines from a text file. The only difference between `read.table` and `read.csv` is that in the later the default separator is a comma. The analogous functions to write data to a text file are called `write.table`, `write.csv`, `writeLines`,...

Let's have a file named *galaxies.dat* which contains:

```
GALAXY morf T.RC3 U-B B-V
NGC1357 Sab 2    0.25 0.87
NGC1832 Sb  4   -0.01 0.63
NGC2276 Sc  5   -0.09 0.52
NGC3245 S0 -2    0.47 0.91
NGC3379 E  -5    0.53 0.96
NGC1234 Sab 3   -0.56 0.84
NGC5678 E  -4    0.45 0.92
```

This file can be read as follows:

```
> gal <- read.table("galaxies.dat",header=TRUE)
```

where the instruction `header=TRUE` specifies that the first line in the file does not contain data but it is a label identifying the contents of every column.

```
> gal                                 # show content of data file
   GALAXY morf T.RC3   U.B  B.V       # 'U-B' and 'B-V' labels have changed!
1 NGC1357  Sab     2  0.25 0.87
2 NGC1832   Sb     4 -0.01 0.63
3 NGC2276   Sc     5 -0.09 0.52
4 NGC3245   S0    -2  0.47 0.91
5 NGC3379    E    -5  0.53 0.96
6 NGC1234  Sab     3 -0.56 0.84
7 NGC5678    E    -4  0.45 0.92
```

The data file is read as a data frame (i.e. a list):

```
> class(gal)
[1] "data.frame"
> names(gal)
[1] "GALAXY" "morf"   "T.RC3"  "U.B"    "B.V"


> gal$morf                            # text chains are read as factors
[1] Sab Sb  Sc  S0  E
Levels: E S0 Sab Sb Sc

> options(stringsAsFactors = FALSE)   # unless default behaviour is disabled
> gal <- read.table("galaxies.dat",header=TRUE)
> gal$morf
[1] "Sab" "Sb"  "Sc"  "S0"  "E"


> tapply(gal$U.B,gal$morf,mean)       # calculate mean colours for every morph. type
    E     S0    Sab     Sb     Sc
0.490  0.470 -0.155 -0.010 -0.090
```

The names of the different fields can be directly accessed (without lists name specification) using their names:

```
> attach(gal)                         # direct access to the list elements
> morf                                #   (it is no longer necessary to use gal$morf,...)
  [1] Sab Sb  Sc  S0  E
  Levels: E S0 Sab Sb Sc

> detach(gal)                         # remove direct access
```

If the data file only contains numbers, information can also be read and assigned to a matrix instead of storing it in a data frame. As an example, if we want to read a file with 3 columns:

```
> a <- matrix(data=scan("numbers.dat",0),ncol=3,byrow=TRUE)
Read 36 items
> a
      [,1]  [,2] [,3]
 [1,]    2  0.25 0.87
 [2,]    4 -0.01 0.63
 [3,]    5 -0.09 0.52
 [4,]   -2  0.47 0.91
 [5,]   -5  0.53 0.96
 [6,]    1  0.45 0.92
 [7,]    3  0.20 0.73
 [8,]   -3  0.51 0.94
 [9,]   -5  0.55 0.96
[10,]   10 -0.22 0.39
[11,]   -1  0.38 0.85
[12,]    5 -0.03 0.63
```

If the number of columns is not specified through `ncol`, all the elements are stored into a one dimensional array:

```
> a1 <- matrix(data=scan("numbers.dat",0))
Read 36 items
> a1
       [,1]
 [1,]  2.00
 [2,]  0.25
 [3,]  0.87
 [4,]  4.00
 [5,] -0.01
 [6,]  0.63
 [7,]  5.00
 [8,] -0.09
 [9,]  0.52
   .     .
   .     .
   .     .
[28,] 10.00
[29,] -0.22
[30,]  0.39
[31,] -1.00
[32,]  0.38
[33,]  0.85
[34,]  5.00
[35,] -0.03
[36,]  0.63
```

If `byrow=TRUE` is omitted the element assignment does not preserve the columns information:

```
> a2 <- matrix(data=scan("numbers.dat",0),ncol=3)
Read 36 items
> a2
       [,1]  [,2]  [,3]
 [1,]  2.00 -5.00 -5.00
 [2,]  0.25  0.53  0.55
 [3,]  0.87  0.96  0.96
 [4,]  4.00  1.00 10.00
 [5,] -0.01  0.45 -0.22
 [6,]  0.63  0.92  0.39
 [7,]  5.00  3.00 -1.00
 [8,] -0.09  0.20  0.38
 [9,]  0.52  0.73  0.85
[10,] -2.00 -3.00  5.00
[11,]  0.47  0.51 -0.03
[12,]  0.91  0.94  0.63
```

Note: Reading large datafiles requires a careful setting of the read.table parameters. Specifying the "colClasses" argument can make the data reading twice as fast while setting the "nrows" argument helps with the memory usage.

6.2 R Example Data

R contains a lot of example data. All the functions and data blocks are stored in packages.

The list of packages currently installed in R can be seen with:

```
> library()
Packages in library '/home/user/R/x86_64-redhat-linux-gnu-library/3.0':

FITSio                  FITS (Flexible Image Transport System) utilities
manipulate              Interactive Plots for RStudio
```

```
plyr                        Tools for splitting, applying and combining  data
rstudio                     Tools and Utilities for RStudio

Packages in library '/usr/lib64/R/library':

base                        The R Base Package
bitops                      Functions for Bitwise operations
boot                        Bootstrap Functions (originally by Angelo Canty for S)
class                       Functions for Classification
...
```

To gather information about a specific package:

```
> library(help=splines)                # show help about the 'splines' package

              Information on package 'splines'

Description:

Package:       splines
Version:       3.0.1
Priority:      base
Imports:       graphics, stats
Title:         Regression Spline Functions and Classes
Author:        Douglas M. Bates <bates@stat.wisc.edu> and William N.
               Venables <Bill.Venables@csiro.au>
Maintainer:    R Core Team <R-core@r-project.org>
Description:   Regression spline functions and classes
...
```

And to load a package and be able to use its functionality:

```
> library(splines)                     # load 'splines' package
```

We can check the data lists that are currently available:

```
> data()
Data sets in package 'datasets':

AirPassengers               Monthly Airline Passenger Numbers 1949-1960
BJsales                     Sales Data with Leading Indicator
BJsales.lead (BJsales)      Sales Data with Leading Indicator
BOD                         Biochemical Oxygen Demand
CO2                         Carbon Dioxide Uptake in Grass Plants
ChickWeight                 Weight versus age of chicks on different diets
DNase                       Elisa assay of DNase
EuStockMarkets              Daily Closing Prices of Major European Stock
...
```

And those that are available in a given package:

```
> data(package="cluster")              # show data available through the package 'cluster'

Data sets in package 'cluster':

agriculture                 European Union Agricultural Workforces
animals                     Attributes of Animals
chorSub                     Subset of C-horizon of Kola Data
flower                      Flower Characteristics
plantTraits                 Plant Species Traits Data
...

> data(animals,package="cluster")      # load 'animals' list from 'cluster' package
```

One of the strongest points in R is that new packages are continuously being generated, including new functionalities. To install a new package:

```
> install.packages("car")              # install 'car' package
Installing package into '/home/ceballos/R/x86_64-redhat-linux-gnu-library/3.0'
...

--- Please select a CRAN mirror for use in this session ---  # ask for a software mirror
```

Once installed we can use it:

```
> library(car)                         # load in memory the functionality defined in 'car'
> data(package="cluster")
Data sets in package 'car':

AMSsurvey                      American Math Society Survey Data
Adler                          Experimenter Expectations
...
```

CHAPTER

SEVEN

GRAPHS

7.1 Graphics package

The "base" 2-D graphics options in R are included in the base package **graphics** (`plot`, `hist`, `boxplot`, etc.):

- `plot(x,y)` and `hist(x)` open a new graphic device if there is not one already open. Defaults are `x11` for Unix, `windows` for Windows and `quartz` for Mac OS X.
- There are many parameters in these functions that can be overridden using the `par` function. They are documented in `?par`

```
> par("pch")                    # default plotting symbol (open circle)
[1] 1
> par(pch=2)                    # change symbol (open triangle)

> par("lty")                    # default line type
[1] "solid"
> par("lwd")                    # default line width
[1] 1
> par("col")                    # default colour
[1] "black"
>  par("mfrow")                 # default number of rows & columns (filled row-wise)
[1] 1 1
> par("mfcol")                  # default number of rows & columns (filled column-wise)
[1] 1 1
```

The plotting process will then be:

1. Set a graphics device

```
> pdf(myfile.pdf,width=10.,height=7.1)        # landscape output plot in PDF format
> postscript(myfile.ps)                       # PS output
> png(myfile.png)                             # PNG
> jpeg(myfile.jpeg)                           # JPG
```

2. Make a plot (see main functions below)

```
> plot(x,y)
```

3. Close device

```
> dev.off()
```

7.2 Important Plotting Functions

plot: this function makes scatterplots or other types of R objects plots

abline: add a straight line to a plot

lines: add connected line segments to a plot

segments: add disconnected line segments to a plot

points: add points to a plot

arrows: add arrows to a plot

polygon: add polygons to a plot

text: add text labels to a plot

title: add labels for *X*,*Y* axes, *title*, *subtitle*, *outer margin*

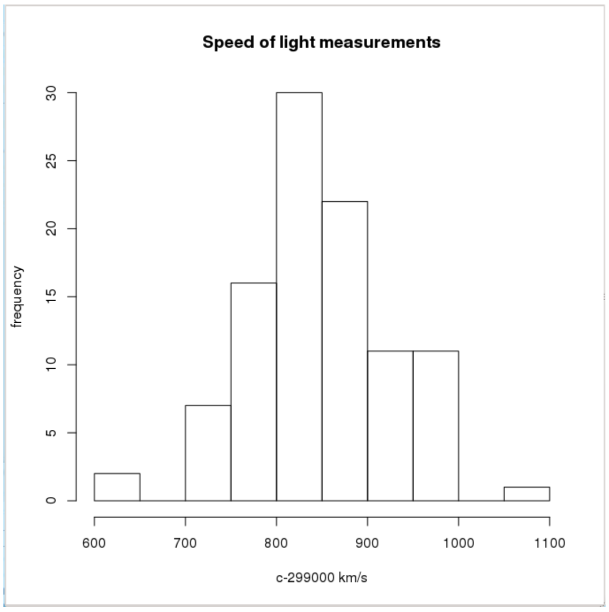
axis: modify axes ticks and axes labels

Let's take the light velocity measures done by Michelson and Morley in their famous experiment. They performed 5 series (*Expt*) with 20 measures each (*Run*):

```
> data(morley)                        # data are in 'base' package loaded by default
> morley
    Expt Run Speed
001    1   1   850
002    1   2   740
003    1   3   900
004    1   4  1070
...
100    5  20   870
```

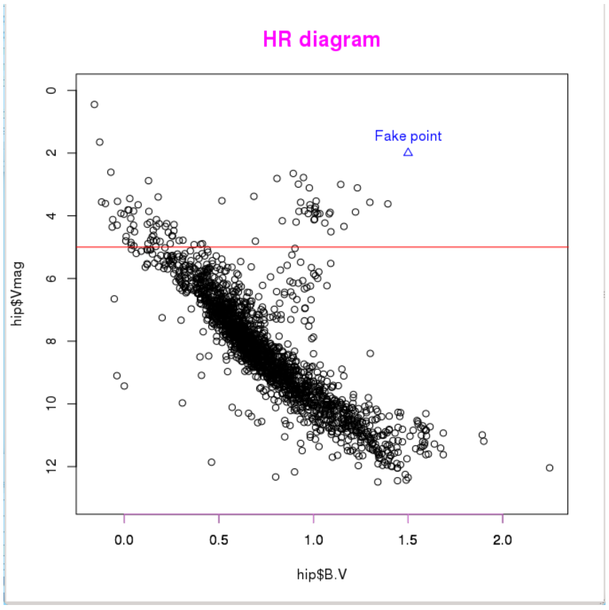
We can generate a histogram:

```
> hist(morley$Speed, main="Speed of light measurements",
+      xlab="c-299000 km/s", ylab="frequency")
```



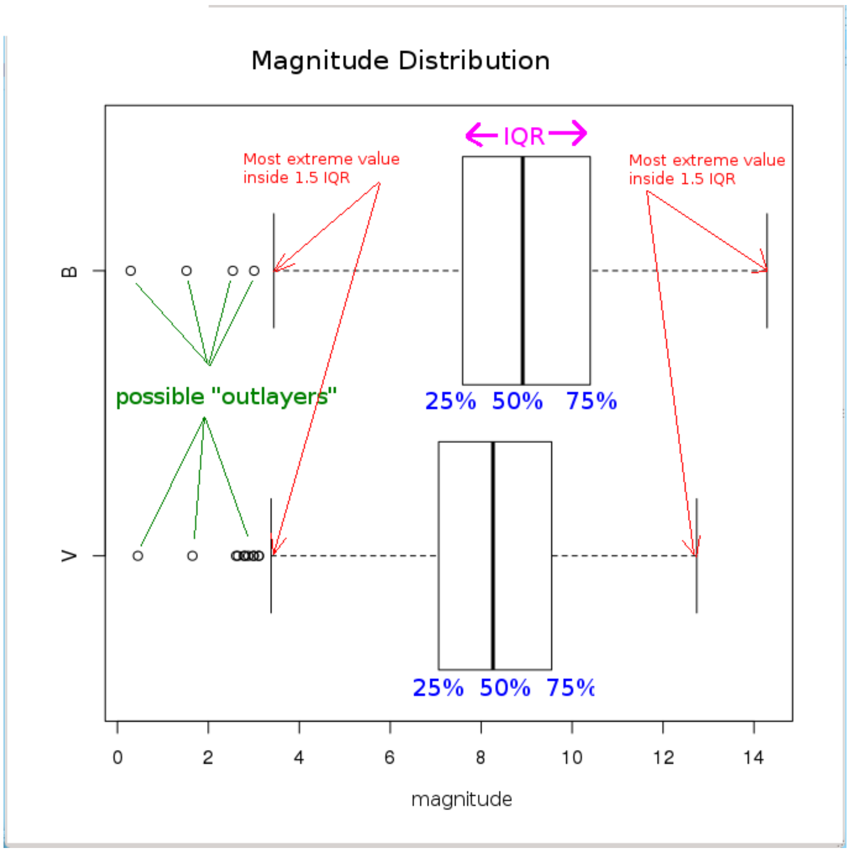
Or a scatter plot:

```
> hip <- read.table("http://www.iiap.res.in/astrostat/tuts/HIP.dat",  # read web file
+                    header=TRUE, fill=TRUE)  # fill=TRUE if some rows have missing values

> names(hip)
[1] "HIP"    "Vmag"   "RA"     "DE"     "Plx"    "pmRA"   "pmDE"   "e_Plx"  "B.V"     # show columns

> plot(hip$B.V,hip$Vmag,ylim=c(13,0))          # scatter plot (black open circle points)
> lines(c(-1,2.5),c(5,5), col="red")
> points(c(1.5),c(2), pch=2, col="blue")
> text(1.5,1.5, "Fake point", col="blue")
> title(main="HR diagram", cex.main=1.5, col.main="magenta")
> axis(1, col="violet")
```



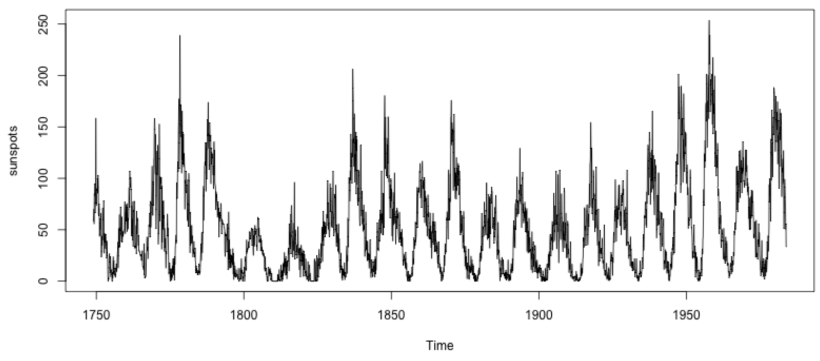
Let's analyse the distribution of star magnitudes we have just loaded. We will first create a new list containing only two components:

```
> hipBmag <- hip$B.V + hip$Vmag                    # calculate B magnitude as "B.V" + "Vmag"
> newlist = list(V = hip$Vmag, B = hipBmag)        # generate a new named list
> names(newlist)                                   # show elements of the new list
[1] "V" "B"
> boxplot(newlist,horizontal=TRUE,                 # create a "box-and-whiskers" plot
+ main="Magnitude Distribution",xlab="magnitude")
```



An example of time series: the monthly mean relative sunspot numbers from 1749 to 1983 (directly available in the package datasets):

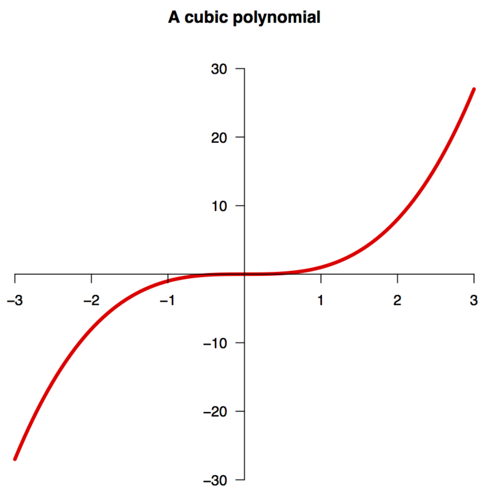
```
> sunspots
       Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
1749  58.0  62.6  70.0  55.7  85.0  83.5  94.8  66.3  75.9  75.5 158.6  85.2
1750  73.3  75.9  89.2  88.3  90.0 100.0  85.4 103.0  91.2  65.7  63.3  75.4
1751  70.0  43.5  45.3  56.4  60.7  50.7  66.3  59.8  23.5  23.2  28.5  44.0
  .     .     .     .     .     .     .     .     .     .     .     .     .
  .     .     .     .     .     .     .     .     .     .     .     .     .
  .     .     .     .     .     .     .     .     .     .     .     .     .
1981 114.0 141.3 135.5 156.4 127.5  90.0 143.8 158.7 167.3 162.4 137.5 150.1
1982 111.2 163.6 153.8 122.0  82.2 110.4 106.1 107.6 118.8  94.7  98.1 127.0
1983  84.3  51.0  66.5  80.7  99.2  91.1  82.2  71.8  50.3  55.8  33.3  33.4

> png("sunspots.png", width=800, height=400)  # define output to PNG file
> plot(sunspots)                              # plot time series
> dev.off()                                   # close PNG file
null device
          1
```



7.3 Simple plots

Although R provides high-level graphics facilities, these tools are built on a set of flexible low-level functions, which sometimes constitute a more flexible approach when creating plots:

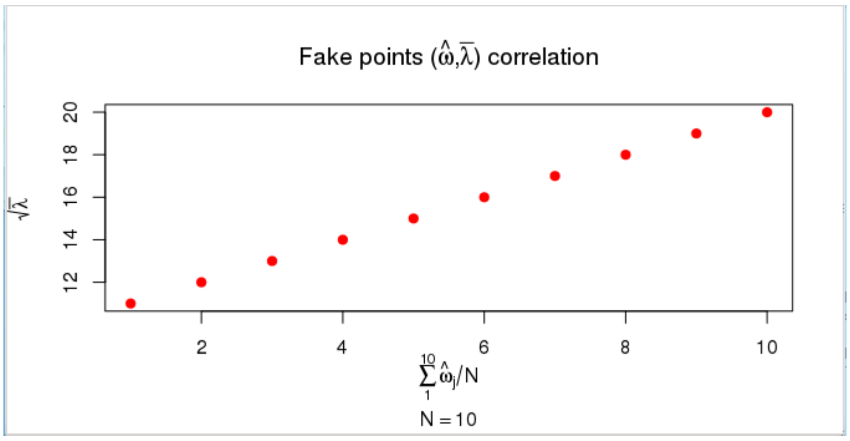
```
# define function to be plotted
> x <- seq(-3,3,length=100)
> y <- x**3

> plot.new()                                  # a new plot is created
> plot.window(xlim=c(-3,3),ylim=c(-30,30))    # set up the world coordinate system
> lines(x,y,col="red",lw=4)                   # plot the curve (red, line width=4)
> axis(1, pos=0, at=c(-3,-2,-1,1,2,3))        # draw X-axis and ticks
> axis(2, pos=0, at=c(-30,-20,-10,10,20,30),
+      las=1)                                 # draw Y-axis and ticks
> title("A cubic polynomial")
```



7.4 Mathematical Annotation

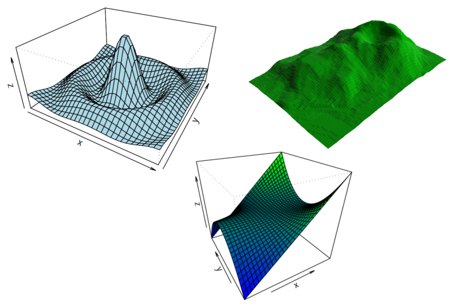
Mathematical symbols can be annotated in R graphs using *expressions* (`expression` function). The possible symbols are listed under `?plotmath`. It is also possible to include computed values in the annotation:

```
> x <- c(1:10)
> y <- c(11:20)
> plot(x, y, main=expression("Fake points (" * hat(omega) * "," * bar(lambda) *
+ ") correlation"), xlab=expression(sum(hat(omega)[j]/N, j=1,10)),
+ ylab=expression(sqrt(bar(lambda))), sub=substitute(N == k, list(k=length(x))),
+ col="red", pch=20, cex=1.5)
```



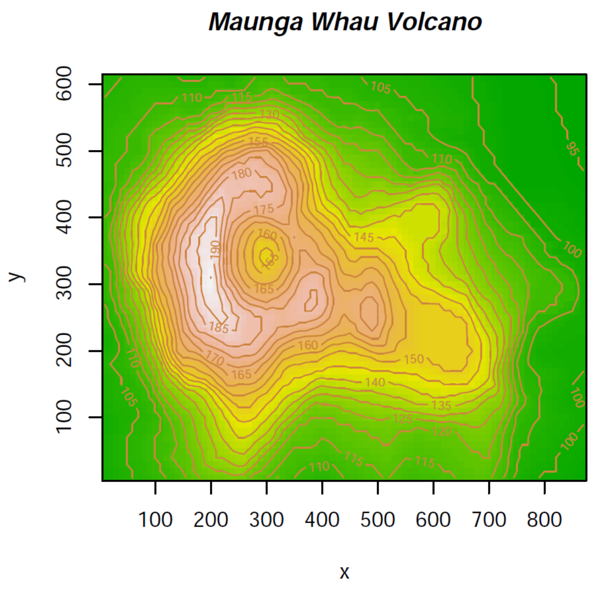
With R you can also make 3D data representations:

```
>  demo(persp)
...
```



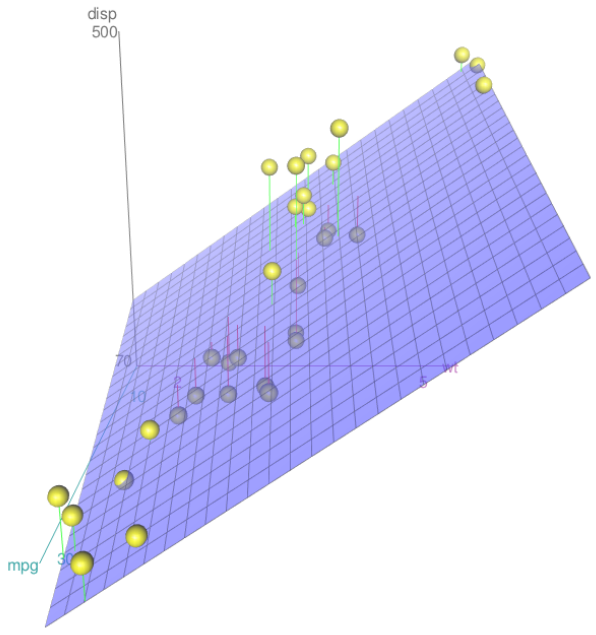
Or image representations:

```
> demo(image)
```



With R you can even make 3D interactive representations:

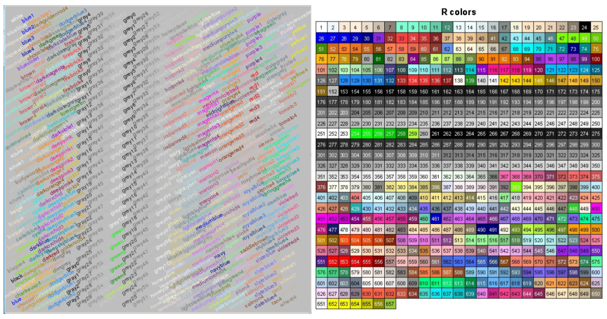
```
> library(car)
> attach(mtcars)
> scatter3d(wt, disp, mpg)
```



7.5 Making use of colours

The colours to use in R graphs can be displayed with:

```
> colors()
  [1] "white"               "aliceblue"             "antiquewhite"
  [4] "antiquewhite1"       "antiquewhite2"         "antiquewhite3"
  ...
  [652] "yellow"              "yellow1"               "yellow2"
  [655] "yellow3"             "yellow4"               "yellowgreen"

> demo(colors)
```



(See the Colors Chart at http://research.stowers-institute.org/efg/R/Color/Chart/index.htm)

However, the use of R colour functions (package `grDevices`) is highly recommendable when plotting coloured graphs.

7.5.1 Colour Palettes

A vector of *n* contiguous colours can be created using the following functions:

rainbow(n, s = 1, v = 1, start = 0, end = max(1, n - 1)/n, alpha = 1)

heat.colors(n, alpha = 1)

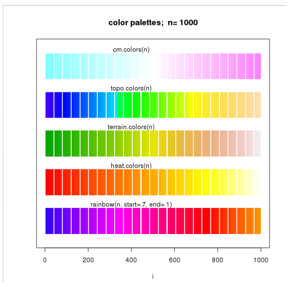
terrain.colors(n, alpha = 1)

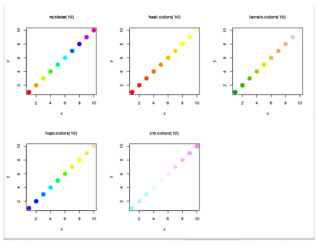
topo.colors(n, alpha = 1)

cm.colors(n, alpha = 1)

```
> x <- c(1:10)
> y <- c(1:10)

> par(mfrow=c(3,2))
> plot(x,y, pch=20, col=rainbow(10), cex=3, main="rainbow(10)", cex.main=1)
> plot(x,y, pch=20, col=heat.colors(10), cex=3, main="heat.colors(10)", cex.main=1)
> plot(x,y, pch=20, col=terrain.colors(10), cex=3, main="terrain.colors(10)", cex.main=1)
> plot(x,y, pch=20, col=topo.colors(10), cex=3, main="topo.colors(10)", cex.main=1)
> plot(x,y, pch=20, col=cm.colors(10), cex=3, main="cm.colors(10)", cex.main=1)
```

The *n* parameter refers to the number of palette colours requested, and *alpha* is the number in [0,1] specified to get transparency (see full documentation in `help(rainbow)`).





7.5.2 Colour Interpolation

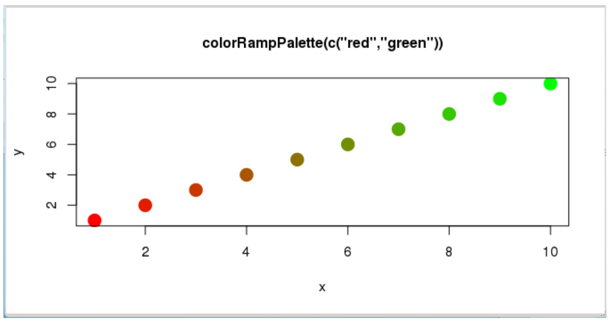
There are functions in R that return functions that interpolate a set of given colours to create new colour palettes and colour ramps:

- **colorRamp**: returns a ‘function’ that maps values between 0 and 1 to colours.

```
> pal <- colorRamp(c("green","blue"))                # define the function

> pal(0)                                              # column 1: RED content
      [,1] [,2] [,3]                                  # column 2: GREEN content
 [1,]    0  255    0                                  # column 3: BLUE content

> pal(0.5)
      [,1]  [,2]  [,3]
[1,]    0 127.5 127.5

> pal(1)                                              # BLUE colour
     [,1] [,2] [,3]
[1,]    0    0  255

> pal(seq(0,1,len=5))
      [,1]   [,2]   [,3]
[1,]    0 255.00   0.00
[2,]    0 191.25  63.75
[3,]    0 127.50 127.50
[4,]    0  63.75 191.25
[5,]    0   0.00 255.00
```

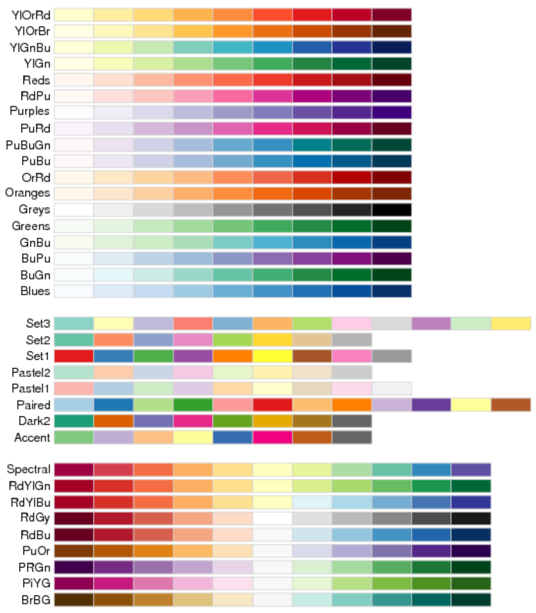
- **colorRampPalette**: returns a function that takes an integer argument and returns that number of colours interpolating the given sequence

```
> x <- c(1:10)
> y <- c(1:10)

> mypal <- colorRampPalette(c("red","green"))

> mypal(10)
 [1] "#FF0000" "#E21C00" "#C63800" "#AA5500" "#8D7100" "#718D00" "#55AA00"
 [8] "#38C600" "#1CE200" "#00FF00"

> plot(x,y, pch=20, col=mypal(10),
+      cex=3, main="colorRampPalette(c(\"red\",\"green\"))", cex.main=1)
```

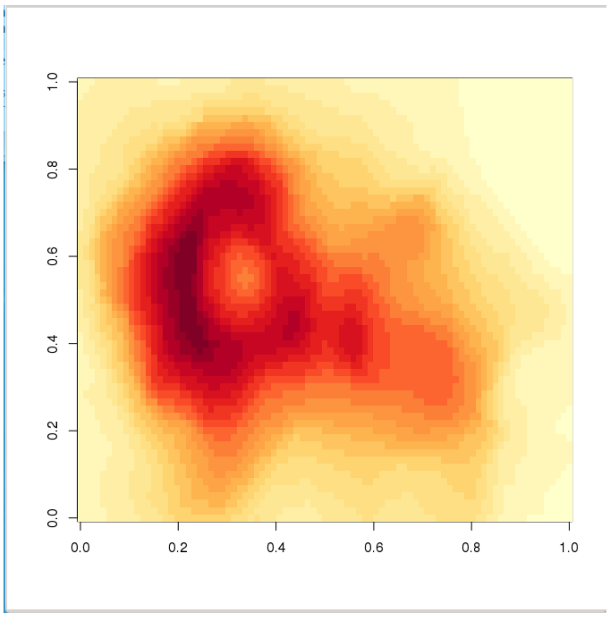


7.5.3 Additional Palettes and colour functions

There is one package installable from CRAN with additional colour palettes (*sequential*, *diverging* and *qualitative* palettes), that can be used with `colorRamp` and `colorRampPalette`: **RColorBrewer**

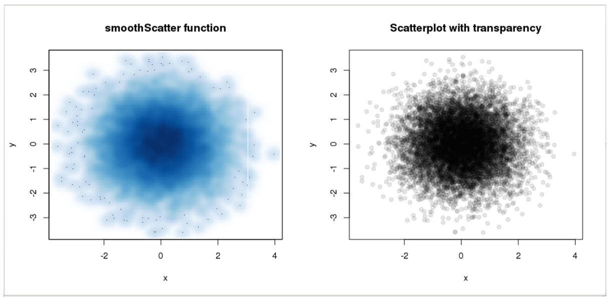


```
> library(RColorBrewer)                        # load library

> colors <- brewer.pal(4, "YlOrRd")            # select 4 of the 9 colours from "YlOrRd" sequence

> colors                                       # show colours selected
[1] "#FFFFB2" "#FECC5C" "#FD8D3C" "#E31A1C"

> mypal <- colorRampPalette(colors)            # create a new (interpolated) palette

> image(volcano, col = mypal(20))              # plot image using 20 colours from new palette
```



When plotting a scatter plot with a lot of points, two options can be used to clarify the plot: `smoothScatter` and *transparency*:

```
> x <- rnorm(10000)
> y <- rnorm(10000)
> par(mfrow=c(1,2))
> smoothScatter(x, y, main="smoothScatter function")
> plot(x,y,col=rgb(0,0,0,0.1), pch=19, main="Scatterplot with transparency")
```



CHAPTER

EIGHT

STATISTICAL TREATMENT

R contains a very comprehensive library with statistical functions, including the most common probability distributions:

Distribution	**R name**	**additional arguments**
beta	`beta`	`shape1, shape2, ncp`
binomial	`binom`	`size, prob`
Cauchy	`cauchy`	`location, scale`
chi-squared	`chisq`	`df, ncp`
exponential	`exp`	`rate`
F	`f`	`df1, df2, ncp`
gamma	`gamma`	`shape, scale`
geometric	`geom`	`prob`
hypergeometric	`hyper`	`m, n, k`
log-normal	`lnorm`	`meanlog, sdlog`
logistic	`logis`	`location, scale`
negative binomial	`nbinom`	`size, prob`
normal	`norm`	`mean, sd`
Poisson	`pois`	`lambda`
Student's t	`t`	`df, ncp`
uniform	`unif`	`min, max`
Weibull	`weibull`	`shape, scale`
Wilcoxon	`wilcox`	`m, n`

8.1 Associated Functions

There are several functions associated to every probability distribution, and they can be accessed adding a prefix to the distribution name:

Prefix	Meaning
d	density function
p	distribution function (cumulative function)
q	inverse of the distribution function (quantile function)
r	random generation of numbers following the probability distribution

The arguments are obviously different for each associated function. For the *Normal Distribution*:

```
> ?Normal

Normal                    package:stats                    R Documentation

The Normal Distribution

Description:

   Density, distribution function, quantile function and random
   generation for the normal distribution with mean equal to 'mean'
   and standard deviation equal to 'sd'.

Usage:


  dnorm(x, mean = 0, sd = 1, log = FALSE)
  pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
  qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
  rnorm(n, mean = 0, sd = 1)

...
```

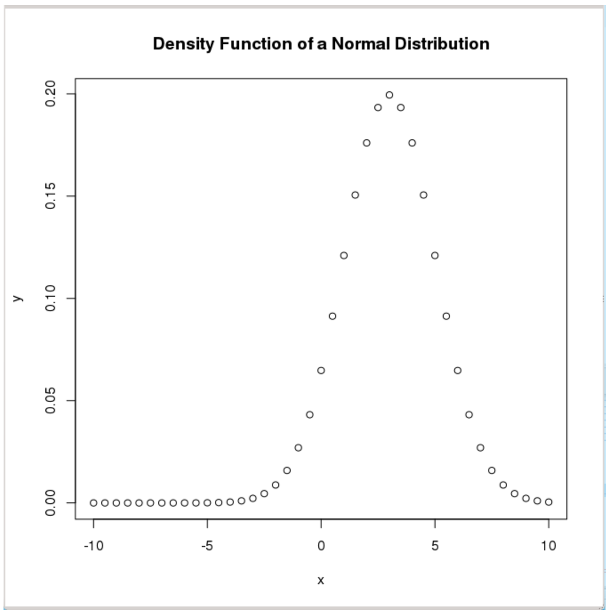
1. dnorm(x, mean = 0, sd = 1, log = FALSE)

It evaluates the density of the normal distribution with mean `mean` and standard deviation `sd` in `x` abscissa. The normal distribution has density

$f(x) = \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

where μ is the mean of the distribution and σ the standard deviation.

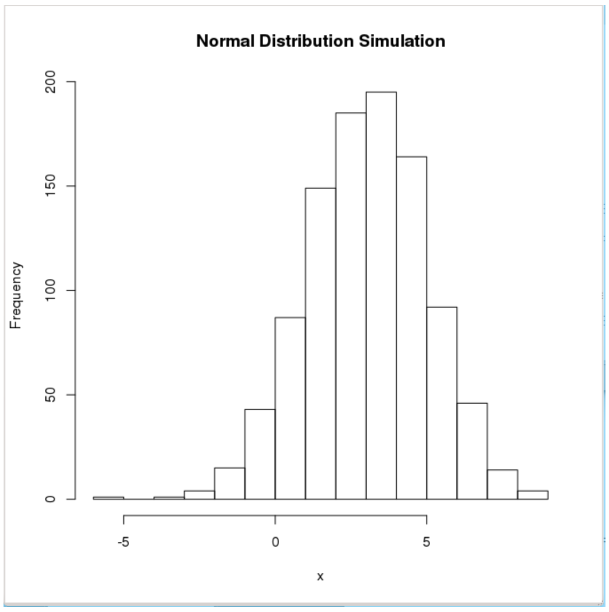
```
> x <- seq(-10,10,by=.5)                      # sequence of numbers
> x
 [1] -10.0  -9.5  -9.0  -8.5  -8.0  -7.5  -7.0  -6.5  -6.0  -5.5  -5.0  -4.5
[13]  -4.0  -3.5  -3.0  -2.5  -2.0  -1.5  -1.0  -0.5   0.0   0.5   1.0   1.5
[25]   2.0   2.5   3.0   3.5   4.0   4.5   5.0   5.5   6.0   6.5   7.0   7.5
[37]   8.0   8.5   9.0   9.5  10.0

> y <- dnorm(x, mean=3, sd=2)                 # Normal distribution with mean=3 and sd=2

> plot(x,y,main="Normal Distribution Example")        # Plot the result
```



2. rnorm(n, mean = 0, sd = 1)

Random sequence of `n` numbers following a normal distribution with mean `mean` and standard deviation `sd`.

```
> x <- rnorm(1000,mean=3,sd=2)                # 1000 random numbers with mean=3 and sd=2
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-3.783   1.694   3.003   3.047   4.436   9.864

> hist(x,main="Normal Distribution Simulation", ylab="Frequency", plot=TRUE)
```

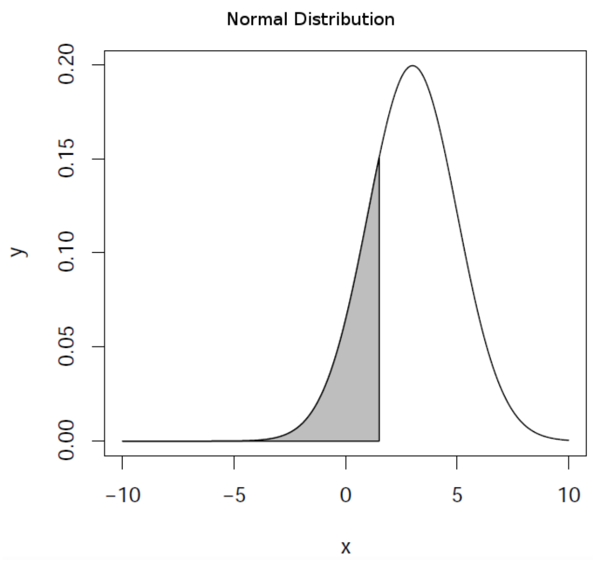


To ensure reproducibility, it is important to set the random number seed when performing simulations:

```
> set.seed(1000)
> rnorm(10)
  [1] -0.44577826 -1.20585657  0.04112631  0.63938841 -0.78655436 -0.38548930
  [7] -0.47586788  0.71975069 -0.01850562 -1.37311776
> rnorm(10)
  [1] -0.98242783 -0.55448870  0.12138119 -0.12087232 -1.33604105  0.17005748
  [7]  0.15507872  0.02493187 -2.04658541  0.21315411
> set.seed(1000)
> rnorm(10)
  [1] -0.44577826 -1.20585657  0.04112631  0.63938841 -0.78655436 -0.38548930
  [7] -0.47586788  0.71975069 -0.01850562 -1.37311776
```

3. pnorm(q, mean = 0, sd = 1, lower.tail = FALSE, log.p = FALSE)

It evaluates the distribution function (area below the probability distribution) for a normal distribution with mean `mean` and standard deviation `sd`. By default, `lower.tail = TRUE` returns the area in the left wing of the distribution ($P[X \leq x]$) and `lower.tail = FALSE` returns the right wing ($P[X > x]$).



```
> pnorm(1.5,mean=3,sd=2)                          # left wing (default)
[1] 0.2266274
```

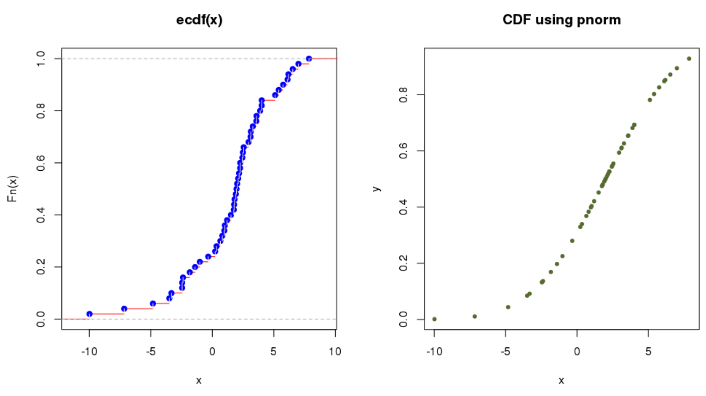
```
> pnorm(1.5,mean=3,sd=2,lower.tail=FALSE)                # right wing
[1] 0.7733726
```

The R object `ecdf(x)` lets us calculate and plot the *Empirical Cumulative Distribution Function* (useful when the cumulative distribution is not known). Let's see with an example how to plot the cumulative function in the case of a normal distribution:

```
> par(mfrow = c(1, 2))                # define 1 row and 2 columns to plot
> x <- rnorm(50, 2, 4)                # random numbers following normal distribution

> plot(ecdf(x),verticals = TRUE, col.points = "blue",
  + col.hor = "red", col.vert = "bisque")      # plot Empirical Cumulative Distribution Function
```

which is equivalent to:

```
> y <- pnorm(x, 2, 4)
> plot(x,y, main="CDF using pnorm",
+       col="darkolivegreen",pch=20) # plot Cumulative Distribution Function using 'pnorm'
```



4. qnorm(p, mean = 0, sd = 1, lower.tail = FALSE, log.p = FALSE)

It evaluates the inverse of the distribution function (the abscissa for an area `p` under the probability distribution) for a normal distribution with mean `mean` and standard deviation `sd`. By default, `lower.tail = TRUE` assumes that the area is that of the left wing of the distribution and `lower.tail = FALSE` assumes that is the right wing area.

```
> qnorm(0.2266274,mean=3,sd=2)                       # left wing (default)
[1] 1.5

> qnorm(0.7733726,mean=3,sd=2,lower.tail=FALSE)       # right wing
[1] 1.5
```

8.2 Common probability distributions

Distribution	Associated Function
Uniform	dunif, punif, qunif, runif
Binomial	dbinom, pbinom, qbinom, rbinom
Poisson	dpois, ppois, qpois, rpois
...	d..., p..., q..., r...
Normal	dnorm, pnorm, qnorm, rnorm
t de Student	dt, pt, qt, rt

chi	dchisq, pchisq, qchisq, rchisq
F de Fisher	df, pf, qf, rf
...	d..., p..., q..., r...

8.3 Example script

Purpose: Estimation of the value of π using random points generated inside a square.

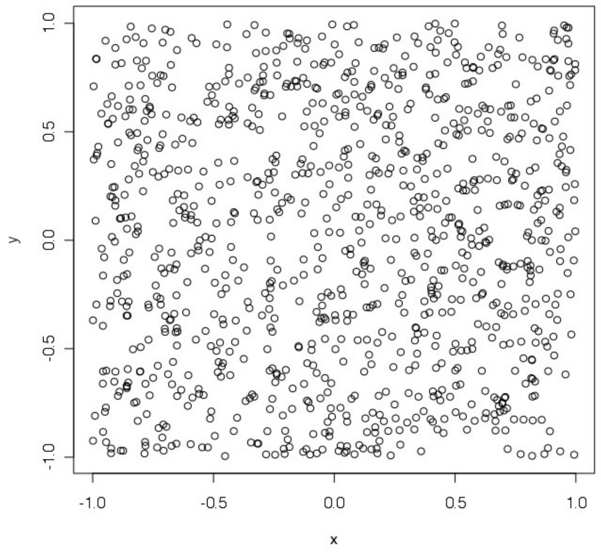
Procedure: Calculate the ratio between the *inner* and *outer* points in a circle with radius equal to 1, inscribed in a square of side equal to 2 (i.e., the circle's diameter is equal to the square's side).

We save the script in a file called `pirandom.R`:

```
# estimate PI by using random numbers
#    A.squ = n = (2*r)²
#    A.cir = n.inside = pi * r²
#
#    pi = n.inside/ r² = 4*n.inside/n
#
pirandom <- function(n)                 # define function
  {
    x <- runif(n,-1,1)                  # random numbers in [-1,1]
    y <- runif(n,-1,1)                  # random numbers in [-1,1]
    plot(x,y) # plot
    r <- sqrt(x*x+y*y)                  # distance to centre
    rinside <- r[r<1]                   # inside circle with r=1?
    n.inside <- length(rinside)
    print(4*n.inside/n)                 # print pi estimation
}
```

The code is executed in R as follows:

```
> source("pirandom.R")                  # load the code (function) in the script
> pirandom(1000)                        # run the function for 1000 points
[1] 3.184                               # 'pi' value estimation
```

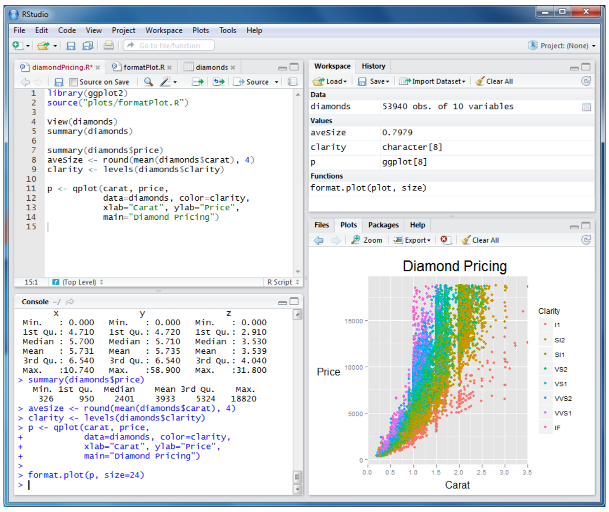


CHAPTER

NINE

RSTUDIO: AN INTEGRATED ENVIRONMENT

There are *Integrated Development Environments* (IDE) that helps in the process of R development. Among them, a very common one is RStudio



```
library(ggplot2)
source("plots/formatPlot.R")

View(diamonds)
summary(diamonds)

summary(diamonds$price)
aveSize <- round(mean(diamonds$carat), 4)
clarity <- levels(diamonds$clarity)

p <- qplot(carat, price,
           data=diamonds, color=clarity,
           xlab="Carat", ylab="Price",
           main="Diamond Pricing")

```

Data

```
diamonds      53940 obs. of 10 variables
```

Values

```
aveSize       0.7979
clarity       character[8]
p             ggplot[8]
```

Functions

```
format.plot(plot, size)
```

```
       x                y                z         
 Min.   : 0.000   Min.   : 0.000   Min.   : 0.000  
 1st Qu.: 4.710   1st Qu.: 4.720   1st Qu.: 2.910  
 Median : 5.700   Median : 5.710   Median : 3.530  
 Mean   : 5.731   Mean   : 5.735   Mean   : 3.539  
 3rd Qu.: 6.540   3rd Qu.: 6.540   3rd Qu.: 4.040  
 Max.   :10.740   Max.   :58.900   Max.   :31.800  
> summary(diamonds$price)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    326     950    2401    3933    5324   18820 
> aveSize <- round(mean(diamonds$carat), 4)
> clarity <- levels(diamonds$clarity)
> p <- qplot(carat, price,
+            data=diamonds, color=clarity,
+            xlab="Carat", ylab="Price",
+            main="Diamond Pricing")
> 
> format.plot(p, size=24)
> 
```

CHAPTER

TEN

BIBLIOGRAPHY AND REFERENCES

10.1 Books

- **R in action**, Robert I. Kabacoff, Manning Publications; 1 edition (August 24, 2011), ISBN-10: 1935182390, ISBN-13: 978-1935182399
- **R for dummies**, Joris Meys, Andrie de Vries, 2012, ISBN-10: 1119962846, ISBN-13: 978-1119962847
- **Beginning R: An Introduction to Statistical Programming**, Larry Pace, Apress; 1 edition (October 17, 2012), ISBN-10: 1430245549, ISBN-13: 978-1430245544
- **Beginning R: The Statistical Programming Language**, Mark Gardener, Wrox; 1 edition (June 5, 2012), ISBN-10: 111816430X, ISBN-13: 978-1118164303
- **Learning RStudio for R Statistical Computing**, Mark P.J. van der Loo and Edwin de Jonge, Packt Publishing (December 24, 2012), ISBN-10: 1782160604, ISBN-13: 978-1782160601

10.2 On-line tutorial and courses

- http://www.cyclismo.org/tutorial/R
- http://tryr.codeschool.com/levels/1/challenges/1
- *Data Analysis*, COURSERA
- *Computing For Data Analysis*, COURSERA
- http://tryr.codeschool.com/

10.3 Center for Astrostatistics

- http://astrostatistics.psu.edu/

10.4 R graphs

- Simple graphics
- CRAN Task View: Graphics

10.5 Blogs

- Quick-R
- R-statistics blog
- R-bloggers
- Revolutions
- R Wiki

CHAPTER

ELEVEN

PDF VERSION

Here you can find a `PDF file` with the contents of this web site.

CHAPTER

TWELVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*