# Programming with Applications in 

## an introduction

Kim-Anh Le Cao & Peter Bailey
University of Queensland

# Contents

# Part I

# Introduction to R

At the end of this introductory part, you should be able to:

- Have a basic understanding of the R philosophy and how it works,
- Be able to find help on any R function and command,
- Use vector-based operations instead of fastidious loops to speed computations,
- Understand the different data structures in R,
- Appreciate the following code:
  `knowledge = apply(theory, 1, sum)`

# Chapter 1

# Introduction

## 1.1 R, what is it?

R is a scripting language for statistical data manipulation and analysis. It has become popular because it is an open software (free) and because more people a contributing to it by submitting *packages* or *libraries*.

What are R and CRAN? (extract from www.http://cran.r-project.org/)

> *R is 'GNU S', a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the R project homepage (www.r-project.org/) for further information.*
>
> *CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. Please use the CRAN mirror nearest to you to minimize network load.*

## 1.2 Help resources on the Internet

There are many excellent resources on R on the Internet. Amongst them:

- The RSeek search engine: http://www.rseek.org/
- The R projects' manuals are available from the R home page: http://www.r-project.org/ (click Manual). They can be sometimes a bit overwhelming.
- I sometimes find Quick-R useful for basics and advances statistics http://www.statmethods.net/
- You can post your questions to the R list server 'r-help' or browse through the old posts. I myself often use this for specific issues/bugs I encounter.

## 1.3 Graphical User Interface (GUI)

Usually, R users submit commands to R by typing in a terminal window. Integrated Development Environments (IDE) are aimed towards programming. Free IDEs have been developed for R:

- RStudio, http://rstudio.org/ **which I recommend**
- StatET is based on Eclipse, http://www.walware.de/goto/statet
- ESS (Emacs Speaks Statistics) is an add-on to Emacs, http://ess.r-project.org/

## 1.4 How is this document organised?

This document will first give some basics about R programming, with a mix of examples and exercises, before introducing specific R functions and techniques for data analysis and data mining.

**Remark 1** *Throughout the document, each R command will be preceeded by the prompt symbol >, which is not part of the code.* ***You should not type the sign> it when you are trying the code yourself. The output will be preceeded by*** [1].
*If the command is too long, it will appear as a continuation symbol + on the following line (you do* **do not need to type the symbol + on your R terminal***).*
*Note that several commands can be typed on the same line using the symbol* `";"`.
*Comments in the R code are preceeded by the # symbol.* ***You do not need to type any comments, unless for your own use.***

**Example**

```
> myvector = c(1,4,2,3)
> myvector   # my comment is: outputs the content of `myvector'. Below is the output:

[1] 1 4 2 3
```

**Remark 2** *The* `c` *stands for concatenate, as here, we are concatenating the values* $1, 4, 2, 3$. *In fact, we are concatenating 4 elementss in one vector.*

## 1.5 R programming

### 1.5.1 Object Oriented Programming

R offers a certain uniformity of access to data: a single function (like `plot()`) can be applied to different types of inputs, which the function processes in the appropriate way. Such a function is called a *generic function* (in C++ it is called 'virtual function'). For example, we will see later that the `plot()` function can be applied to a list of numbers, or to the output of a regression analysis, which represent various aspects of the analysis. What it means from a user's perspective is that there are fewer commands to remember!

### 1.5.2 Functional programming

R avoids explicit iterations (loops). This is a very important concept to remember: instead of coding loops, we will exploit R's functional features, which let us express iterative behaviour implicitly. The code is run more efficiently and it makes a huge timing difference when dealing with large data sets such as those generated by high-throughput experiments in Molecular Biology. For instance, to compute the mean of a vector containing 6 values `v`$= [1, 1, 2, 3, 4, 4]$,

```
> v = c(1, 1, 2, 3, 4, 4)
> v         # outputs the content of the vector 'v'

[1] 1 1 2 3 4 4

> mean(v)   # outputs the mean of the elements in the vector

[1] 2.5
```

The command `mean(v)` computes the mean of the vector `v`, instead of of summing of the elements one by one and dividing the sum by 6 like this (*you do not need to run the command line, but try appreciate the length of this code*):

```
> sum.vect = 0  # initialise for first element
> for(i in 1:6){   # create a 'for' loop
+   sum.vect = sum.vect + v[i]
+ }
> mean.vect = sum.vect/6
> mean.vect
```

```
[1] 2.5
```

Another example of R functional features is the following:

```
>  sum(v)/length(v)  # sum of all elements of v / size of v
```

```
[1] 2.5
```

## 1.6 Getting started

### 1.6.1 Running R

R operates in two modes: *interactive* and *batch*. In this course we will use interactive mode, which is the most typical where we type in commands, R displays the results, then we type in more commands etc. The batch mode does not require interaction with the user. It is useful for production jobs as the process is automated. Keep the batch mode in mind when you have large data set to process with high-performance computing and you know that the job will run for a long time.

### 1.6.2 Starting R

R has the notion of your current *working directory*. This will be the directory from which you launch R, if you are using Linux or Mac. In Windows, it might be your Document folder. Check your current directory by typing the command:

```
> getwd()
```

```
[1] "/Users/k.lecao/Teaching/STAT7174/Prac/prac-R"
```

If you are using an IDE, you can change your working directory.
Otherwise you can use the command

```
> setwd("~/Teaching/STAT7174/Prac/prac-R/")
```

to set up your working directory.

## 1.7 Getting help

Besides the numerous resources available in Internet, as mentioned in Section 1.2, R is self-documenting.

### 1.7.1 The `help()` function

To get online help, invoke the function `help()`. For example on the `boxplot` function:

```
> help(boxplot)
```

or, alternatively with the shortcut `?`:

```
> ?boxplot
```

Special characters must be quoted, for example on the `<` operator:

```
> ?"<"
```

or with the `for` in the loops:

```
> ?"for"
```

Much work has gone into making R self-documenting and each of the help entries comes with examples. The function `example()` will run these examples in front of your very own eyes: However, because of this self-documenting freedom, you can sometimes find the explanations of some functions very obscure ... this is, from my point of view, one of the major drawback of R.

### 1.7.2 When you do not really know what you are looking for

The `help.search()` function does a Google-search type through R's documentation. For example we want to generate a box-and-whiskers plot: `help.search("box")`
or using the shortcut `"??"`

```
> ??"box"
```

will produce a response containing
`graphics::boxplot  Box Plots`
which means we should have a look at the function `boxplot()` in the package `graphics`.

### 1.7.3 Help for other topics

We can also get some help about an entire package:

```
> help(package = graphics)
```

and also for some general topics:

```
> ?Math
```

```
Help on topic 'Math' was found in the following packages:

  Package               Library
  methods               /Library/Frameworks/R.framework/Versions/2.15/Resources/library
  base                  /Library/Frameworks/R.framework/Resources/library


Using the first match ...
```

## 1.8 Functions in R

R programming consist of writing functions. A function is a group of instructions that takes inputs (the *arguments*), uses them to compute other values and returns a result.
A function is always called with arguments in round brackets. With no arguments, we will still need to write the round brackets, otherwise it is the actual code of the function that appears:

```
> round(3.4)
```

```
[1] 3
```

has the argument "3.4" and returns the nearest integer 3.

```
> ls()
```

```
[1] "i"         "mean.vect" "myvector"  "sum.vect"  "v"
```

lists the objects in the working environment, while

```
> ls
```

outputs the whole R code of the function!

Many functions in R use *default arguments*. For example the function `round()` takes two arguments: `x` and `digits = 0`. The latter is a *default argument*. We will come back to it later in the course but it is important when using any R function to be aware of the values of these default arguments (detailed in the help documentation, see Section 1.7).
**Tip**: When using Rstudio, "tabbing" inside the parentheses function will display the arguments.

## 1.9 Packages

One of the major strengths in R are the thousands of user-written packages available through the Comprehensive R Archive Network (CRAN, http://www.r-project.org/. As of March 2011, there were 4,300 packages available from the CRAN. A package is a collection of functions, which allow specialized statistical techniques, graphical devices, import/export capabilities, reporting tools, etc. These packages are developed primarily in R, and sometimes in Java, C and Fortran.
Some packages are loaded automatically when you start R (such as the `base` package). But to save memory and time, R does not load *all* available packages automatically.

*Remark.* The term *library* is often used in place of package in the R community.

**Load a package from your hard drive.** You can load a package that is in your R installation but not loaded into memory yet using the `library()` function. For example for the package `MASS`:

```
> library(MASS)
```

Any function within this package will now be ready to be used.

### 1.9.1 Download a package from the Web.

The package you want to use may not be in your R installation. You can install it automatically from the CRAN repository with the `install.package()` function, or, using a GUI or IDE by clicking on "Install package" (choose your nearest mirror).

### 1.9.2 Install a package manually.

You may not have the rights on your machine, or you may want to make modifications to an existing package to create your own package. You can install a package by hand using some batch command in Linux or Mac. This is not in the scope of this course but be aware of the possibility.

## 1.10 Shutting down

R can save all or part of a session as a record of what you did, to an output file. If you answer "yes" to the question "Save workspace image?" when you quit R by typing `q()`.

R will save all the objects you created in that session in a file named `.RData` located in your R directory. It will restore the objects the objects in the next session if you load that same workspace: `load(".RData")` or through the IDE you are using.

**Tip**: sometimes save my objects in a middle of a session, just in case my computer crashes with the command `save.image()`.

You can also save your R workspace with a specific name, for example: `save.image('prac0.RData')` and load it next session with `load('prac0.RData')`.

## 1.11 One important tip

Once the R software has shut down (or worse, crashed), the workspace might have been saved beforehand, so you will keep the R objects in memory. However, all the command lines will be gone. It is best to use a **text editor**, preferably the one embedded in the R software (file -> New -> R script) that you can save as an '.R' file (for example 'practical0.R').
First type your command line in your text editor, then copy paste it in the R console to excute (or use the 'apple key + enter' in a mac, or 'Ctrl + enter' on Windows with the cursor pointing on the command line, or use the Run button). **It might sound tedious at the start to use a command editor but it will be useful later**, especially for debugging and typing high level R commands.

Note: the `.RHistory` can be saved using the command line `savehistory()`, but it will display every possible (possibly wrong) command lines that have been typed during the session.

# Chapter 2

# Data structure

## 2.1 Introduction to data structure

R has a variety of data structures.

**Vector.** The **vector** type is the fundamental data type in R. The elements of a vector must all have the same *mode* (data type) of the following type:

- **Scalars** are individual numbers, they actually are one-element vectors

```
> x = 10  # example of scalar
> x

[1] 10

> mode(x)

[1] "numeric"
```

- **Character strings** are single-element vectors of mode "character"

```
> y = "hello"  # example of character
> y

[1] "hello"

> mode(y)

[1] "character"
```

**Matrix.** A **matrix** is a rectangular array of numbers (technically a vector with a number of rows and a number of colums)

```
> # first example
> M = matrix(data = 0, nrow = 2, ncol = 3)
> M   # a matrix with 0 values, 2 rows and 3 columns

     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0

> # second example
> M = matrix(c(1,2,3,4,5,6), nrow = 2, ncol  =3)
> M
```

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

Matrices are indexed using double scripting which start from 1:

```
> M[1,2]   # outputs the element from the first row and the second column

[1] 3
```

We can extract submatrices from a matrix

```
> M[1,]  # extracts the first row

[1] 1 3 5

> M[,2]  # extracts the second column

[1] 3 4
```

**Arrays.** Arrays are matrices with more than 2 dimensions. In Section **??** we will give an example of three-dimensional arrays.

**Lists.** lists contain items of different data types (similar to a C struct in C++).

```
> x = list(u=2, v = "hello")
> x

$u
[1] 2


$v
[1] "hello"
```

Elements in a list can be accessed via the $ sign:

```
> x$u

[1] 2

> x$v

[1] "hello"
```

**Data frame.** A data frame is a data set that contains data of different modes (i.e. different types). A data frame is a list with each component of a list being a vector corresponding to a column in the data.

```
> D = data.frame(list(name = c('patient 1', 'patient 2', 'patient 3'),
+                     age = c(34, 21, 56), tumor = c('yes', 'no', 'yes')))
> D

       name age tumor
1 patient 1  34   yes
2 patient 2  21    no
3 patient 3  56   yes

> D$name

[1] patient 1 patient 2 patient 3
Levels: patient 1 patient 2 patient 3
```

Typically, data frames are created by reading in a data set from a file or a database.

**Classes.** Since R is an object oriented language, objects are instances of **classes**. This concept is a bit more abstract than data types and we will come back to it later in the course.

**Remark 3** *You will notice in the following code that we can use the sign '<-' instead of '='.* `<-` *is the standard assignment operator in R (i.e. I assign to my object a, the value 10:* `a <= 10`*). The sign '=' can also be used.*

## 2.2 Scalar

```
> 2+2
> exp(10)
> a = log(2)
> b <- cos(10)  #Remark: '<-' is equivalent to '='. In fact,
>               # '->' is also possible with the variable name following the symbol.
> a+b
> a
> b
> 2==3
> b = 2 < 3
> ls()
> rm(a)
> ls()
> a="foo"
```

**Exercise 1** . *Type each of the above command lines one by one and comment your script if needed. Then answer the questions below.*

1. *Identify the mode of each scalar: integer, real, boolean (logical), character.*
2. *What does the* `ls()` *function do?*
3. *What does the* `rm()` *function do?*

## 2.3 Vector

- These following commands illustrate how to create a vector, either by indicating the values in `c()`, or with a sequence `seq()` or with a repetition `rep()`.
- Elements in a vector are of the same type (numeric, character)
- To extract an element from a vector we use the operator `[ ]`.
- You can also name an element of a vector (see example below with `f`).
- The notation `NA` stands for (*Not Available*) and indicates missing values

```
> d = c(2,3,5,8,4,6)
> is.vector(d)
> c(2,5,"foo")
> 1:10
> seq(from=1,to=20,by=2)
> seq(1,20,by=5)
> seq(1,20,length=5)
> rep(5,times=10)
```

```
> rep(c(1,2),3)
> rep(c(1,2),each=3)
> e = rep(1,10)
> d[2]
> d[2:3]
> d[-(1:2)]
> d[3]=NA
> d
> summary(d)
> is.na(d)
> help(NA)
> any(is.na(d))
> all(is.na(d))
> f = c(a=12,b=26,c=32,d=41)
> f
> names(f)
> f["a"]
> names(f)=c("a1","a2","a3","a4")
> f >30                      # (1)
> f[f > 30]                  # (2)
> which(f>30)
> f[2] = 22
> f+100
> f+d                        # (3)
> cos(f)
> length(f)
> sort(d)
```

**Exercise 2** . *Type each of the above command lines one by one and comment your script if needed. Then answer the questions below.*

1. *Which are the different arguments that we can use in the function* `seq()`*? Give some examples.*
2. *Which are the different arguments that we can use in the function* `rep()`*? Give some examples.*
3. *Describe the function* `unique()`*? Give an example.*
4. *What is the use of the functions* `any()` *and* `all()`*?*
5. *What is the difference between the two commands in (2).*
6. *What is happening with (3)* `f+d` *?*

**Part of first Quizz, exercise 1** *This is an exercise as part of the quizz.*

1. *Create a vector object called '*`marks`*', which records the marks of the students:[1 pt]*

| *Martin* | *Joey* | *Stephany* | *Allan* | *Sophie* | *Lila* |
|---|---|---|---|---|---|
| *80* | *75* | *15* | *55* | *60* | *95* |

2. *Check the size of the vector. [1 pt]*

3. *Don't forget to name each element of the vector with the students name. [1 pt]*

4. *Who are the students who pass the course ($> 50$)? [1 pt]*

5. *What is the average mark of the students? What is the maximum and the minimum mark? [1 pt]*

6. *Who has the maximum mark (use R commands)? [1 pt]*

## 2.4 Matrix

- Similar to vectors, elements of a matrix are of the same type.
- Some arguments in a function can be shortened (the number of columns `ncol` can be written `nc` for example).

```
> A = matrix(1:15,ncol=5)
> A
> B = matrix(1:15,nc=5,byrow=T)
> B2=B
> B2[1,1]="foo"
> B2
> cbind(A,B)
> rbind(A,B)
> A[1,3]
> A[,2]
> B[2,]
> A[1:3,2:4]
> g = seq(0,1,length=20)
> C = matrix(g,nrow=4)
> C[C[,1] > 0.1,]                # (1)
> D = matrix(runif(16),nc=4)
> D > 0.5                        # (2)
> D[D[,1] > 0.5,2]               # (2')
> A+B
> A*B
> A[1:2,1:2] %*% B[1:2,1:3]
> apply(A,2,sum)                # (3)
> apply(D,1,max)
```

**Exercise 3** . *Type each of the above command lines one by one and comment your script if needed. Then answer the questions below.*

1. *What is the difference between the two commands in (2) and (2').*

2. *Explain (3)?*

**Part of first Quizz, exercise 2** . *Answer the questions below as part of the exercise for Quizz 1.*

1. *We record the pulse rate of 20 persons who have either a drink of caffeinated or decaffeinated cola. Create the following matrix '`pulse.rate`': [1 pt]*

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| *Caffeinated* | *87* | *92* | *96* | *97* | *78* | *78* | *94* | *90* | *80* | *96* |
| *Decaffeinated* | *100* | *75* | *97* | *81* | *91* | *93* | *78* | *76* | *81* | *76* |

2. *Using the command* `rownames`*, name the rows of your matrix. [1 pt]*
3. *Give the average pulse rate for each type of drink. [1 pt]*
4. *In the Caffeinated group, how many persons have a pulse rate* $\geq 90$*? [1 pt]*

# Part II

# Graphical outputs in R

At the end of this introductory part, you should be able to:

- Understand the basics of using R's base graphics packages,
- Be able to handle data matrices, use subscripts and extract submatrices,
- Be able to handle factors,
- Be able to handle a small microarray data set,
- Be able to represent data with graphics in R

# Chapter 3

# Introduction to graphics in R

## 3.1 Introduction

R has a very rich set of graphics facilities. For example you can browse the R Graph Gallery: http://gallery.r-enthusiasts.com/
We will first have a look at the foundational function for creating graphcs using `plot()`. We will then explore how to build a graph, adding lines and attaching a legend. We will deal with plots with two variables, a single sample, multivariate plots and special plots for particular purposes.

### 3.1.1 Saving graph to file

There are different ways of saving a graph to a file

- Standard R GUI, right-click on the graph and save
- Rstudio: click on the buttons on top of the plots
- In a Linux environment:

```
> pdf('mygraph.pdf')
> plot(cars)
> dev.off()

null device
          1
```

Note that in the latter case, several format are available, such as `jpeg(), bmp(), png(), tiff, postcript()` ... (see the help of these functions). The command `dev.off()` enables to close the R graphic device.

## 3.2 `plot()`, the workhorse for R Base graphics

### 3.2.1 First example

Suppose we have two vectors $x = [1, 2, 3]$ and $y = [1, 2, 4]$, interpreted as a set of pairs in the $(x, y)$ plane.

```
> x = c(1,2,3)
> y = c(1,2,4)
> plot(x, y)
```

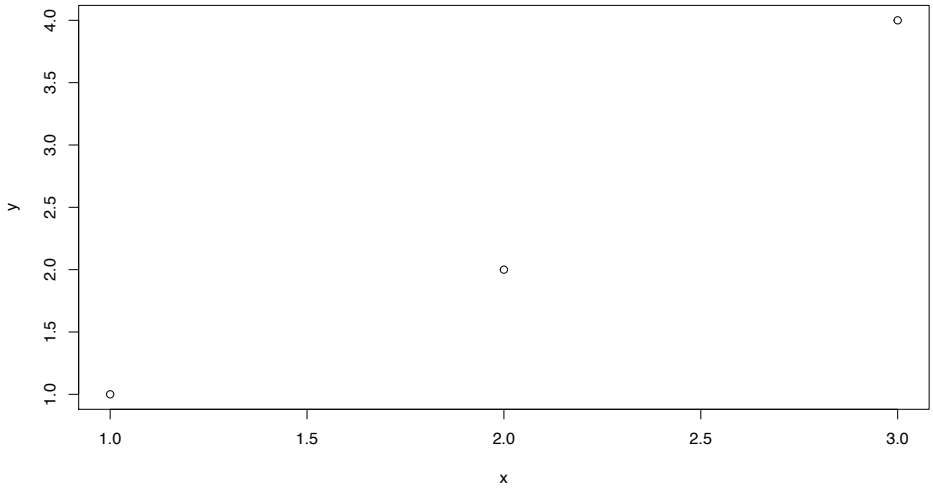



Figure 3.1: First simple example of scatterplot.

### 3.2.2 Changing the shape of points and their color

The empty circles can be changed with the argument `pch`, and the color by the argument `col` (see Figure 3.4(a)):

```
> plot(x, y, pch = 16, col = 'blue')
```

All point character symbols are represented in Figure 3.2. The function `colors()` lists all the possible 657 colors.

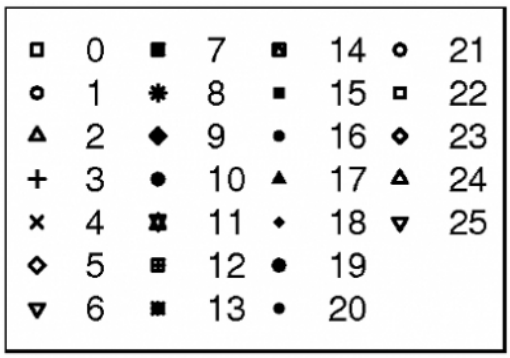



Figure 3.2: Point character symbols.

### 3.2.3 Other useful arguments

The ranges of the $x$ and $y$ axes can be set up with a vector with 2 elements (start and finish) using the arguments `xlim` and `ylim`. The labels of the axes can also be modified (see Figure 3.4(b)):

```
> plot(x, y, pch = 16, col = 'blue', xlim = c(0,4), ylim = c(0,5),
+      xlab = 'the label of x', ylab = 'the label of y')
```

### 3.2.4 Adding lines

After the call to `plot()`, the call to the function `abline()` will add a line to the current graph. Any straight vertical, horizontal lines or lines with intercept and slope can be added (see also Figures 3.3 and 3.4(a)).

```
> plot(x, y, pch = 16, col = 'blue', xlim = c(0,4), ylim = c(0,5),
+      xlab = 'the label of x', ylab = 'the label of y')
> abline(h = 2.5, col = 'green')  #horizontal line
> abline(v = 2, col = 'red')      #vertical line
> abline(2,1)                     #line fitted on the equation y = 2 + 1 * x
```

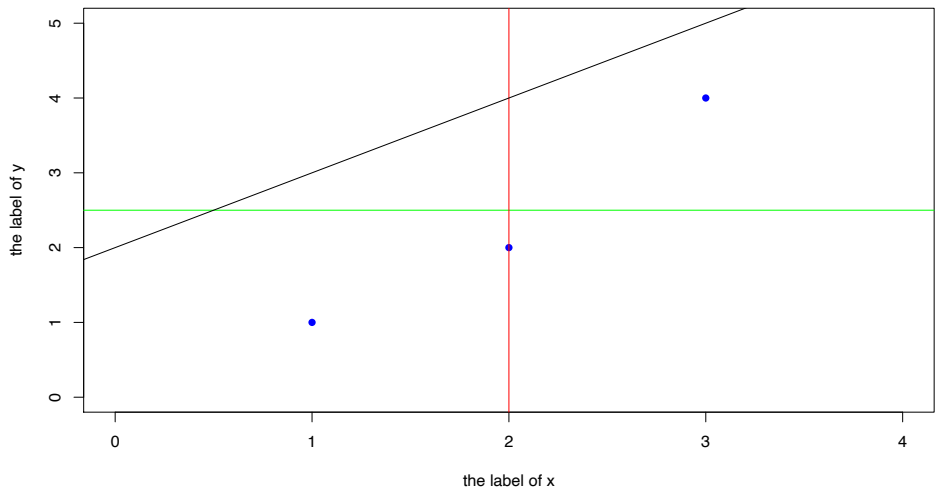



Figure 3.3: Example adding lines.

### 3.2.5 Adding a legend

Legend can be added to a graph (see also `example(legend)`, amd Figure 3.4(d)).

```
> plot(x, y, pch = c(16, 17, 18), col = c('blue', 'red', 'green'))
> legend('bottomright', pch = c(16, 17, 18), col = c('blue', 'red', 'green'),
+        legend = c('small', 'medium', 'large'))
```

### 3.2.6 Dividing the graphic device window into panels

This is useful to represent several graphics on the same window, using the command `par(mfrow=c(nr,nc))` where `nr` is the number of rows and `nc` the number of columns to be divided. Titles can also be added.

```
> # divide into panels
> par(mfrow=c(2,2))  # here: 2 rows and 2 columns
> # --- plot 1
> plot(x, y, pch = 16, col = 'blue')
> title(main = '(a)')
> # --- plot 2
> plot(x, y, pch = 16, col = 'blue', xlim = c(0,4), ylim = c(0,5),
+      xlab = 'the label of x', ylab = 'the label of y')
```

```
> title(main = '(b)')
> # --- plot 3
> plot(x, y, pch = 16, col = 'blue', xlim = c(0,4), ylim = c(0,5),
+      xlab = 'the label of x', ylab = 'the label of y')
> abline(h = 2.5, col = 'green')  #horizontal line
> abline(v = 1, col = 'red')  #vertical line
> title(main = '(c)')
> # --- plot 4
> plot(x, y, pch = c(16, 17, 18), col = c('blue', 'red', 'green'))
> legend('bottomright', pch = c(16, 17, 18), col = c('blue', 'red', 'green'),
+        legend = c('small', 'medium', 'large'))
> title(main = '(d)')
> par(mfrow=c(1,1)) # pack to 1 panel
```

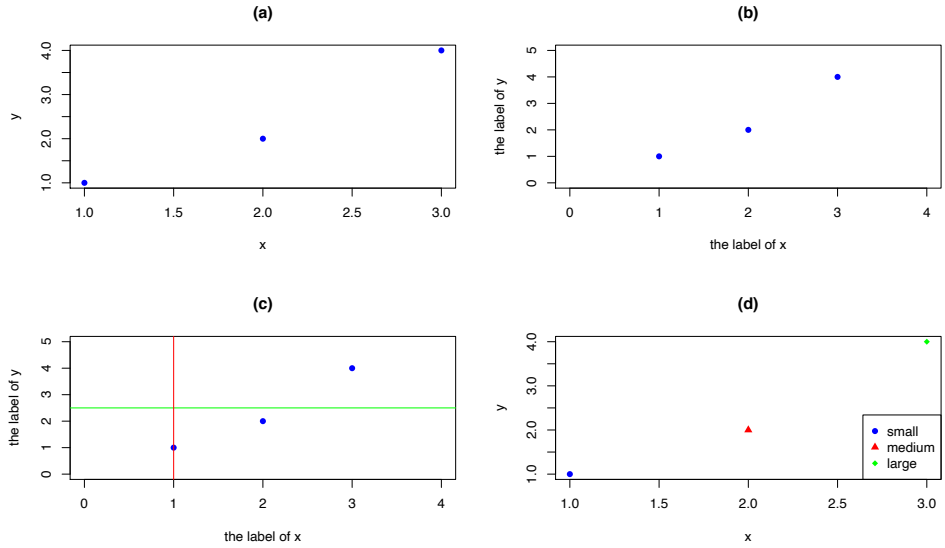



Figure 3.4: Examples of plots. (a): changing shape and color, (b): changing range of axes and labels, (c): adding lines, (d): adding legend.

# Chapter 4

# A first microarray example

At the end of this practical you should be able:

1. to extract gene expression values from a matrix;
2. to generate and use factors to group categorical data;
3. to use the functions apply and tapply to calculate descriptive statistics on matrices.

## 4.1 Load the data

The gene expression data collected by Golub et al. (1999) are among the most classical in bioinformatics. The data relates to Leukemia. We will concentrate today on a subset of this data, refered to herein as 'golub100', which is provided in the R data file `golub.RData`. `golub100` is a matrix consisting of 100 genes (rows) and their respective expression values in 38 leukemia patients (columns). Twenty seven patients are diagnosed as having acute lymphoblastic leukemia (ALL) and eleven as having acute myeloid leukemia (AML).

```
> # 1. Empty the previous workspace
> # (the following removes (almost) everything in the working environment.
> # You will get no warning, so don't do this unless you are really sure).=
> rm(list = ls())  # if you want to can delete all previous workspace
> # 2. Then load the data (saved into a Folder called 'Data')
> load('Data/Golub.RData')
> ls()           # lists current objects in the current workspace
> dim(golub100)  #check the dimension of the data
> head(golub100) # outputs the first elements of the data
```

## 4.2 The CCND3 Cyclin D3 gene in Golub study

We shall first concentrate on the expression values of the gene "CCND3 Cyclin D3". Golub et al. demonstrated that the gene cyclin D3 is differentially expressed between ALL and AML tumours.

How can we extract expression values from the `golub100` matrix corresponding to those of Cyclin D3 (CCND3)?

**Exercise 4** *Comment the following R code.*

```
> ccnd3 <- grep("CCND3", rownames(golub100))
> golub100[ccnd3,]
> golub100[95,]
```

**Exercise 5** *Extract values for the gene CCND3 for patients diagnosed with ALL only.*

## 4.3 Comparing ALL patients to AML patients using `factors`

Objects of type `factor` can be used to group categorical data. For example, we can use an object of type `factor` to categorize the `golub100` data on the basis of tumour class i.e. AML versus AML. The tumour class is given by the column names of the matrix `golub100`. We can use the column names in the function `as.factor()` to create the object `gol.factor` as follows.

```
> ?factor
> colnames(golub100)
> gol.factor <- as.factor(colnames(golub100))
> gol.factor
> summary(gol.factor)
```

Once we have created `gol.factor` we can extract expression data specific to the "AML" and "ALL" tumour classes as follows.

```
> golub.aml <- golub100[, gol.factor=="AML"]
> golub.all <- golub100[, gol.factor=="ALL"]
```

## 4.4 Handling matrices

How do we perform calculations on matrices? For instance, how would you calculate the sum, mean, median or standard deviation for the AML or ALL samples. To perform computations on matrices, R provides the helper functions `apply()` and `tapply()`.

```
> ?apply
> mean.aml <- apply(golub100[, gol.factor=="AML"], 1, mean)
> mean.amlvsall <- tapply(golub100[ccnd3,], gol.factor, mean)
```

In the first example we have used the apply function to calculate the mean of all genes from tumours diagnosed as AML. The *numerical argument* ***1*** in the apply function designates that we are intereted in calculting the mean on the *rows* or genes of the matrix. If we wanted to calculate the mean expression values for the AML samples (the *columns*) we would use the *numerical argument* ***2***.

In the second example we have used the `tapply()` function to calculate the mean expression values for the gene CCND3 in tumours diagnosed as either AML or ALL *given a factor*. Note that we have used the `gol.factor` object as the second argument to `tapply()`.

**Exercise 6** *Give the median, sd and IQR of the ALL expression values using the* ***`apply()`*** *function. (Hint: ?median(), ?sd(), ?IQR()).*
*Give the median, sd and IQR of the gene expression values for the gene CCND3 for both the ALL and AML tumours using the* ***`tapply()`*** *function.*

# Chapter 5

# Plotting functions

A few essential methods are given to display and visualize data. It quickly answers questions like: Does the distribution of my data resemble that of a bell-shaped curve? Are there differences between gene expression values taken from two groups of patients?

At the end of this pratical you should be able to display a series of descriptive statistics using the R's base graphics packages.

In the following, we will use the `golub100` data from chapter 4.

## 5.1 Scatter plots

```
> plot(golub100[1,], golub100[92,])
> plot(golub100[1,], golub100[18,])
```

**Exercise 7** *What do you observe? Are the expression levels of the genes positively or negatively correlated?*

## 5.2 Strip chart

```
> stripchart(golub100[ccnd3,] ~ gol.factor, method="jitter", vertical=TRUE)
```

**Exercise 8** *What do you observe?*

## 5.3 Histogram

```
> par(mfrow=c(2,2))          # divide graphics window into 2 rows and 2 columns
> hist(golub100[ccnd3, ])
> hist(golub100[ccnd3, gol.factor=="ALL"])
> hist(golub100[ccnd3, gol.factor=="AML"])
> par(mfrow=c(1,1))         # graphics window back to normal
```

**Exercise 9** *Comment on these histograms.*

## 5.4 Boxplot (box-and-whiskers plot)

A popular method to display data is by drawing a box around the first and the third quartile (a bold line segment for the median), and the smaller line segments (whiskers) for the smallest and the largest data values. Such a data display is known as a box-and-whisker plot.

```
> boxplot(golub100[ccnd3,])
> boxplot(golub100[ccnd3,] ~ gol.factor)
```

**Exercise 10** *What do you observe? What can you say about the expression levels of CCND3?*

**Part of first Assignment, exercise 1** . *Box-and-wiskers plot of persons of Golub et al. (1999) data.*

1. *Use* `boxplot(golub100)` *to produce a box-and-whiskers plot for each column (patient). Make a screen shot to save it in a word processor. Describe what you see. Are the medians of similar size? Is the inter quartile range more or less equal. Are there outliers?*
2. *Compute the mean and medians of the patients. What do you observe?*

## 5.5 Quantile-Quantile plot

A method to visualize the distribution of gene expression values is by the so-called quantile-quantile (Q-Q) plot. In such a plot the quantiles of the gene expression values are displayed against the corresponding quantiles of the normal (bell-shaped) distribution. A straight line is added representing points which correspond exactly to the quantiles of the normal distribution. By observing the extent in which the points appear on the line, it can be evaluated to what degree the data are normally distributed. That is, the closer the gene expression values appear to the line, the more likely it is that the data are normally distributed.

```
> qqnorm(golub100[ccnd3, gol.factor=="ALL"])
> qqline(golub100[ccnd3, gol.factor=="ALL"])
```

**Exercise 11** *What do you observe? What conclusions can you make regarding the normality of the expression values?*

# Chapter 6

# Hypothesis testing

At the end of this practical you should be able to:

1. Test a null hypothesis against an alternative hypothesis using the in-built function `t.test()`;
2. Be able to apply the Chi-Squared test on categorical data; and
3. Be able to apply an analysis of variance (ANOVA) to a set of data

**Remark 4** *A hypothesis is one-sided if it proposes that a parameter is greater than some value or less than some value; it is two-sided if it simply says the parameter is not equal to some value.*

## 6.1 The T-distribution, a reminder

This is the R code to obtain Figure 6.1:

```
> df=6-1
> f<-function(x){dt(x, df)}
> plot(f,-4,4,xlab="x-axis",ylab="T Density dt(x)")
> ci <- c(qt(0.025, df), qt(0.975, df))
> x<-seq(-4,ci[1],0.01)
> y<-seq(ci[2],4,0.01)
> polygon(c(ci[1],x,ci[1]), c(0,0,f(x)), col="red")
> polygon(c(ci[2],y,ci[2]), c(f(y),0,0), col="red")
> arrows(-3, 0.25,-3, 0.05)
> text(-3, 0.3, "Rejection Region")
> arrows(3, 0.25,3, 0.05)
> text(3, 0.3, "Rejection Region")
> arrows(0, 0.15,0, 0.05)
> text(0, 0.2, "Acceptance Region")
```

## 6.2 One Sample t-test

```
> boxplot(golub100[ccnd3,], main = 'CCND3 for all patients', col = 'grey')
```

The box-and-whiskers plot in Figure 6.2 suggests that the ALL gene expression values of CCND3 Cyclin D3 are positive. We would like to test this.

**Exercise 12** *Formulate the null and alternative hypothesis that we would like to test.*

```
> t.test(golub100[ccnd3, gol.factor=="ALL"], mu=0, alternative=c("greater"))
```

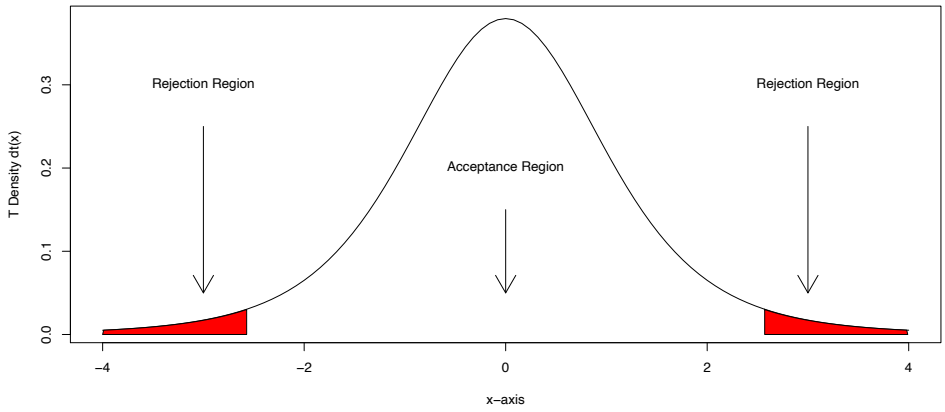



Figure 6.1: Acceptance and rejection regions in a t-test distribution.

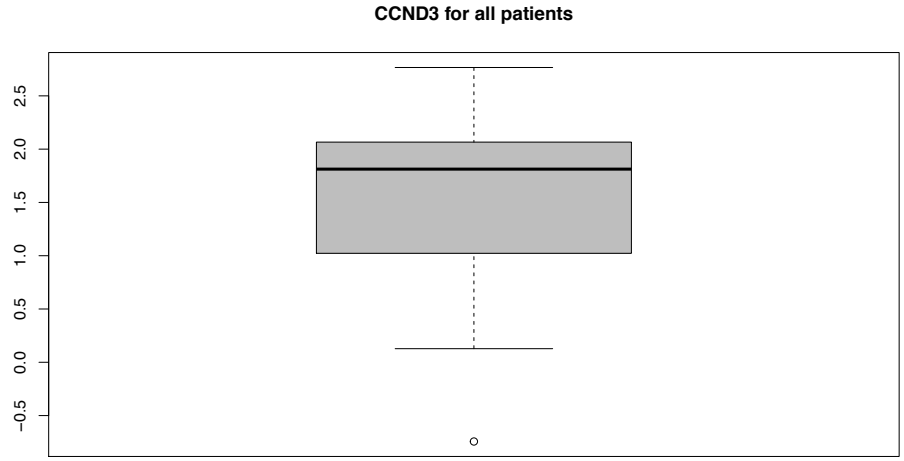



Figure 6.2: Boxplot of CCDN3 for all patients.

The option `mu` provides a number indicating the true value of the mean (or difference in means if you are performing a two-sample test) under the null hypothesis. The option `alternative` is a character string specifying the alternative hypothesis, and must be one of the following: `"two.sided"` (which is the *default*), `"greater"` or `"less"` depending on whether the alternative hypothesis is that the mean is different than, greater than or less than `mu`, respectively.

**Exercise 13** *Explain the R output and draw conclusions.*

## 6.3 Two Sample t-test with unequal variance

Now, we display the boxplot for each group of patients:

```
> boxplot(golub100[ccnd3,]~gol.factor, main = 'CCND3 for the two groups of patients', col = 'grey')
```

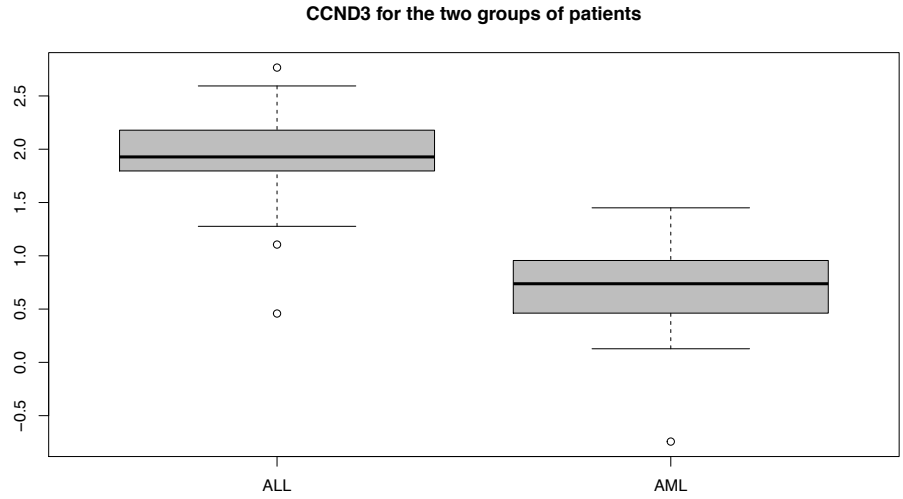



Figure 6.3: Boxplot of CCDN3 for the two groups of patients.

**Exercise 14** *The Welsh's t-test.*

- *What do these boxplots suggest regarding the difference in the population means?*
- *Formulate the null and alternative hypotheses to be tested.*
- *Suggest a statistical test to test your hypothesis. Assess whether the test should be run with equal or unequal variances (argument* `var.equal` *set to* `TRUE` *or* `FALSE`*).*
- *Once your run the test, specify the rejection region and draw conclusions.*

```
> t.test(golub100[ccnd3,] ~ gol.factor, var.equal=FALSE)
```

## 6.4 Two sample t-test with equal variances

The null hypothesis for gene CCND3 Cyclin D3 that the mean of the ALL differs from that of AML patients can be tested by the two-sample t-test using the argument `var.equal=TRUE`.

```
> t.test(golub100[ccnd3,] ~ gol.factor, var.equal=TRUE)
```

**Exercise 15** *State the null hypothesis of this test, the statistical test used, the rejection region and draw conclusions.*

## 6.5 F-test on equal variances

The null hypothesis for gene CCND3 Cyclin D3 that the variance of the ALL patients equals that of the AML patients can be tested by the built-in-function `var.test()`, as follows.

```
> var.test(golub100[ccnd3,] ~ gol.factor)
```

**Exercise 16** *State the null hypothesis of this test, the statistical test used, the rejection region and draw conclusions.*

## 6.6 Normality tests

To test the hypothesis that the ALL gene expression values of CCND3 Cyclin D3 from Golub et al. (1999) are normally distributed, the Shapiro-Wilk test can be used as follows.

```
> shapiro.test(golub100[ccnd3, gol.factor=="ALL"])
```

**Exercise 17** *State the null hypothesis of this test, the statistical test used, the rejection region and draw conclusions.*

## 6.7 $\chi^2$ test on Breast Cancer data

It has been hypothesised that breast cancer in women is caused in part by events that occur between the age of menarche (i.e. the age when menstruation begins) and the age at first child birth. Specifically, it has been hypthesised that the risk of breast cancer increases as the length of this time interval increases. An international study was set up to test this hypothesis.
Women with at least one birth were arbitarily divided into 2 categories:

1. women whose age at first birth was $\leq 29$; and
2. women whose age at first birth was $\geq 30$.

Control subjects were chosen from women of comparable age who were in hospital at the same time as the subjects with breast cancer. However, the control subjects did not have breast cancer. All women were asked about their age at first birth.
The following results were found among women with at least one birth: 683 out of 3220 (21.2%) with breast cancer (case women) and 1498 out of 10245 (14.6%) without breast cancer (control women) had an age at first birth $\geq 30$. How can we assess whether the difference observed between the case women and the control women is significant or simply due to chance?

$2 \times 2$ Contingency tables and Chi-Square test:

```
> tab <- matrix(c(683, 2537, 1498, 8747), 2, byrow=TRUE)
> dimnames(tab) <- list(group=c("case", "control"), age=c(">=30", "<=29"))
> chisq.test(tab)
```

**Exercise 18** *State the null hypothesis of this test, the statistical test used, the rejection region and draw conclusions.*

## 6.8 One-way analysis of variance

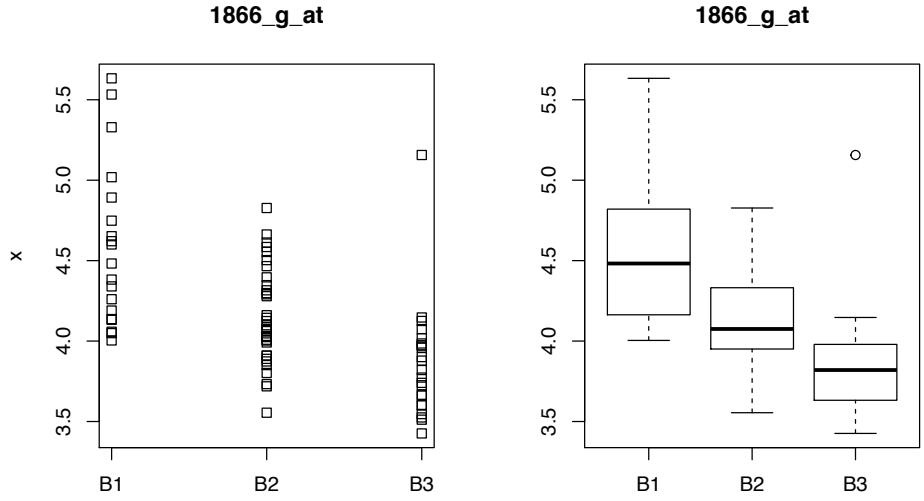
We have seen that the $t$-test can be used to discover genes with different means in the population with respect to two groups of patients. In some cases, however, when there are more than two groups of patients, the question arises how many genes are differentially expressed between group means (experimental effect)? A technique making this possible is an analysis of variance. It is frequently applied in bioinformatics.

For this example we use the matrix ALLB123 which contains the expression values for B-cell ALL patients in stages B1, B2, and B3 of the disease. We are interested in the expression of the SKI-like oncogene which is identified by probe name '1866_g_at'.

```
> x <- ALLB123["1866_g_at",]
> bt.factor
> summary(bt.factor)
```

Some useful graphical outputs can be plotted:

```
> par(mfrow=c(1,2))  #divide graphics window into 1 row 2 columns
> # 1. a strichart
> stripchart(x ~ bt.factor, vertical=T)
> title(main = '1866_g_at')
> # 2. a boxplot
> boxplot(x ~ bt.factor, main = '1866_g_at')
> par(mfrow=c(1,1)) # graphics window back to normal
```




The built-in-function `aov()` can be used to perform the analysis of variance.

```
> test = aov(x ~ bt.factor)
> summary(test)
> # summary(test) outputs a list with two elements, the first element [[1]] is the output table.
> # to extract the p-value:
> summary(test)[[1]]["Pr(>F)"][1]
```

**Exercise 19** *Comment on the outputs above, state the null hypothesis of this test, the statistical test used, the rejection region and draw conclusions.*

**Part of first Assignment, exercise 2** *Hypothesis testing 1/2.*

1. *Gene selection. We are interested in the following list of candidate genes for the Golub study (you will use the* `golub100` *data from the practical):*

```
list.gene = c(
  "LYZ Lysozyme", "CTSD Cathepsin D (lysosomal aspartyl protease)",
  "Clone 23721 mRNA sequence", "Neuromedin B mRNA",
  "DHPS Deoxyhypusine synthase",
  "GB DEF = (lambda) DNA for immunoglobin light chain",
  "Leukotriene C4 synthase (LTC4S) gene", "KIAA0102 gene" ,
  "Non-lens beta gamma-crystallin like protein (AIM1) mRNA, partial cds",
  "CD24 signal transducer mRNA and 3' region")
```

   (a) *For each of these genes, perform a two-sample t-test values for which the ALL mean is greater than the AML mean (test with unequal variances). You will formulate the null and alternative hypotheses, the test statistic used (and the distribution of the test statistic, including the number of degrees of freedom), the rejection region and draw your conclusion. (Advice: you can create a vector called* `p.value` *which will store the p-values associated to the tests, and name* `p.value` *using the function* `names`*).*

   (b) *Report amongst these genes those that have a p-value $< 0.01$. We will refer to these genes as 'differentially expressed' genes.*

   (c) *Illustrate these results by plotting these differentially expressed genes using boxplots. Interpret the boxplots.*

**Part of first Assignment, exercise 3** *Hypothesis testing 2/2.*

1. *Gene CD33. Use the* `grep()` *function to find the index of the important gene CD33 among the rownames of* `golub100`*. For each test below, formulate the null and alternative hypotheses, the test statistic used (its value and the distribution of the test statistic, including the number of degrees of freedom), the rejection region and draw your conclusion:*

   (a) *Test the normality of the ALL and AML expression values*[1]*.*

   (b) *Test for the equality of variances.*

   (c) *Test for the equality of the means by an appropriate t-test.*

   (d) *Is the experimental effect strong?*

# Chapter 7

# Clustering and visualisation

## 7.1 Distances and linkage methods

We will first have a look at a simple example to compare the effect of different linkage methods. The `foodstuff` data set measures the amount of energy, protein, calcium and iron in different types of food. First, we need to read the data from a .txt format (you will need to point your `R` working directory to the right location of the data file):

```
> food <- read.table("foodstuffs.txt", header=T, row.names= 1)
```

Since each variable (energy, protein etc...) does not use the same unit, we decide to scale the data by dividing each variable column by its own standard deviation:

```
> # Let's first center and scale the data
> food.std = scale(food, center = TRUE, scale = TRUE)
```

In order to obtain a cluster, we first need to compute the distance (or measure of similarity), for example with a Euclidian distance:

```
> # Calculating pairwise Euclidean distances between the (standardized) objects:
> dist.food <- dist(food.std)
```

Now we can use the `hclust()` function specifying a linkage method and plot the dendrogram:

```
> # Example with single linkage:
> food.single.link <- hclust(dist.food, method='single')
> # Plotting the single linkage dendrogram:
> plclust(food.single.link,  ylab="Distance")
```

**Exercise 20** *Compare the different dendrograms obtained using the single, the complete and the average linkage. Comment.*

In order to extract the different clusters, we use the `cutree()` function (note that in this type of unsupervised analysis, we need to choose the number of clusters):

```
> # Cutting the complete-linkage dendrogram to form k=2 clusters here:
> cut.2 <- cutree(food.single.link, k=2)
> cut.2     # printing the "clustering vector"

BB HR BR BS BC CB CC BH LL LS HS PR PS BT VC FB AR AC TC HF MB MC PF SC DC UC
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  2  1  1  1  1  1  1  1  1
RC
 1
```

```
> # Suppose we preferred a 5-cluster solution:
> cut.5 <- cutree(food.single.link, k=5)
> # Equivalently, in this case:
> cut.5 <- cutree(food.single.link, h=3.5)
> # h specifies the height at which the dendrogram should be cut
> cut.5   # printing the "clustering vector"

BB HR BR BS BC CB CC BH LL LS HS PR PS BT VC FB AR AC TC HF MB MC PF SC DC UC
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
RC
 1
```

We can also visualise the clusters via a scatterplot matrix:

```
> pairs(food, panel=function(x,y) text(x,y,cut.5))
```

**Exercise 21** *Comment on the characteristics of each type of cluster.*

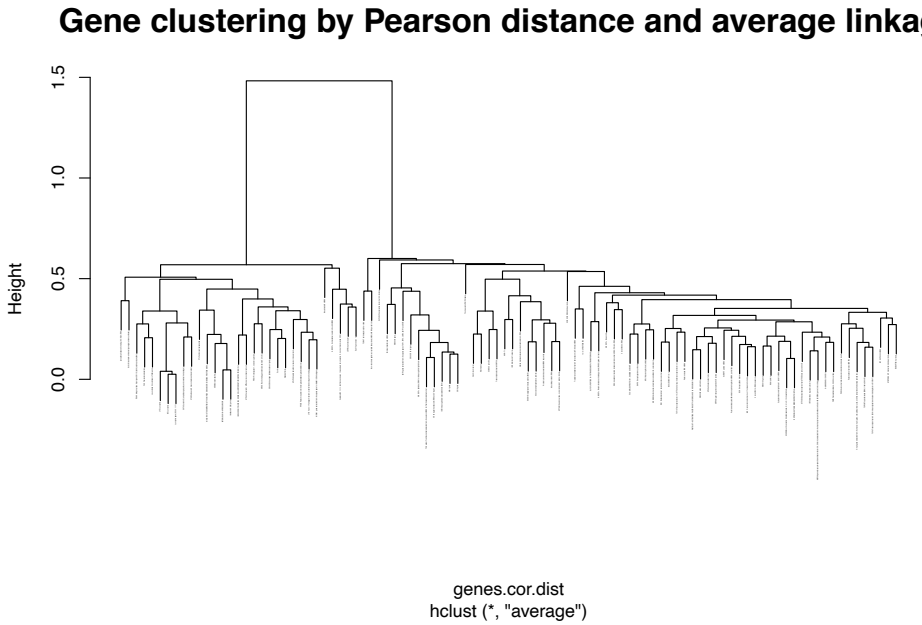
## 7.2 Hierarchical clustering

Clustering is a common analysis performed for DNA microarray data. The most often performed clustering method is hierarchical clustering which typically takes the form of a heatmap. As an example, we will apply hierarchical clustering to the selection of genes provided in the data `golub100`.
We will first compute the Pearson correlation between the genes. Note that we must operate on the transpose of the matrix because the R function `cor()` operates on the columns.

```
> genes.cor <- cor(t(golub100), use="pairwise.complete.obs", method="pearson")
> genes.cor[1:10,1:10]
```

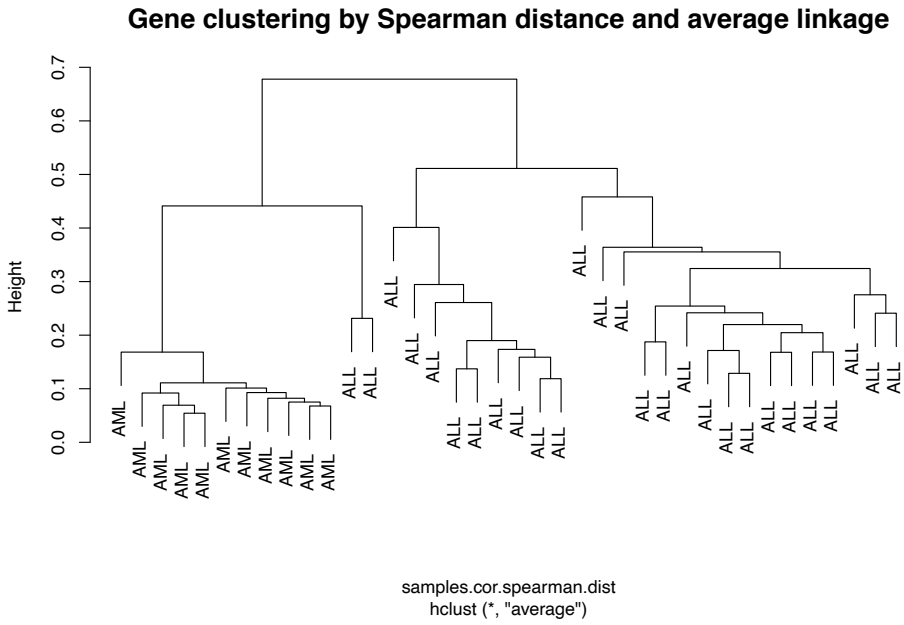
The Pearson correlation coefficient is a similarity metric, values of which vary from -1 (perfect anticorrelation) to +1 (perfect correlation), see Lectures Chap.1. Pearson's correlation can be transformed into a distance metric by subtracting from 1. The Pearson distance would then vary from 0 (perfect correlation) to 2 (perfect anti-correlation). Hierarchical clustering of the Pearson distance metric can be achieved by the `hclust()` function. We will use the `"average"` linkage as an agglomeration rule.

```
> genes.cor.dist <- as.dist(1-genes.cor)
> genes.tree <- hclust(genes.cor.dist, method='average')
> plot(genes.tree, main="Gene clustering by Pearson distance and average linkage", xlab=NULL,
+      cex=0.1, cex.main=1.8)
```




Next we compute the correlations between samples by using the Spearman rank as the metric.

```
> samples.cor.spearman <- cor(golub100,use="pairwise.complete.obs", method="spearman")
> samples.cor.spearman.dist <- as.dist(1-samples.cor.spearman)
> samples.tree <- hclust(samples.cor.spearman.dist, method='average')
> plot(samples.tree, main="Gene clustering by Spearman distance and average linkage",
+      xlab=NULL, cex.main=1.5)
```



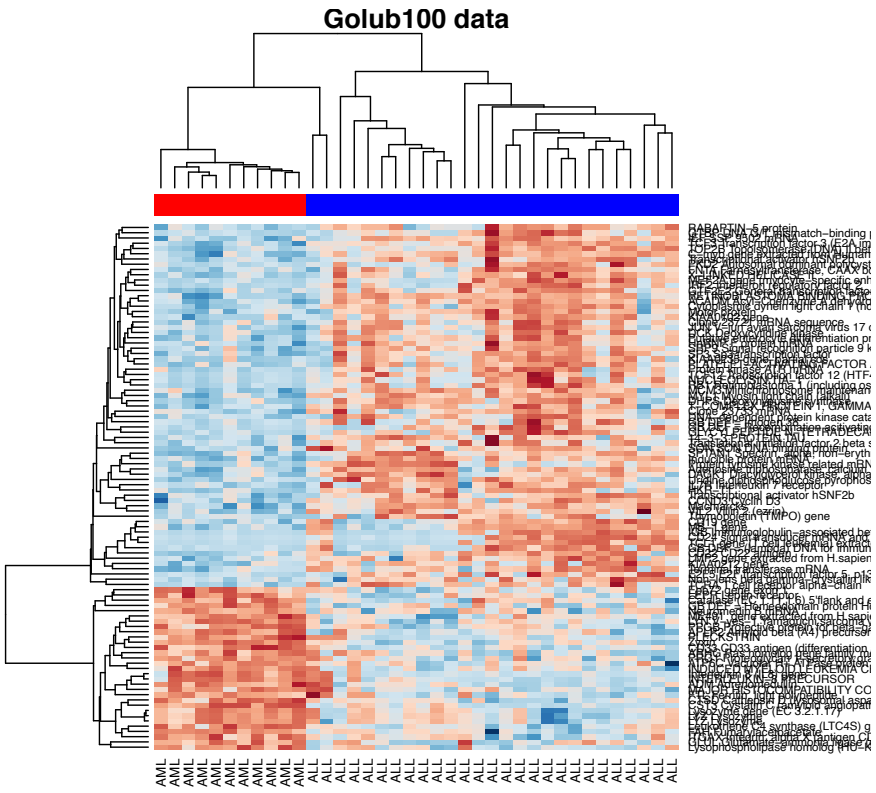

**Exercise 22** *What other agglomeration methods may be used with the hclust method? Interpret the sample clustering obtained above.*

We can now generate a heatmap for the clustered genes and samples using the `heatmap()` function. By default `heatmap()` automatically performs hierarchical clustering on the rows and columns, using Euclidean

distance and complete linkage. We can, however, use the parameters RowV and ColV to impose our own clustering results. The rows and columns are reordered so as to match the branches of the trees.

We set the colors for the heatmap with the `colorRampPalette()` function from the package `RColorBrewer`. We have selected colours ranging from red to blue. We reverse the selected colours so that red corresponds to high values and blue corresponds to low values.

```
> library("RColorBrewer")
> # 1. set the colors for the gene expression data
> col <- colorRampPalette(brewer.pal(10, "RdBu"))(256)
> col <- rev(col)   # reverse colors as indicated above
> # 2. set the colors for the samples
> patientcolors <- ifelse(gol.factor =='AML', 'red', 'blue')
> heatmap(golub100,
+         scale="row",
+         col=col,
+         Rowv=as.dendrogram(genes.tree),
+         Colv=as.dendrogram(samples.tree),
+         main="Golub100 data",
+         ColSideColors=patientcolors
+         )
```




**Remark 5** *You can also try a simpler command line with less options:*

```
> heatmap(golub100, scale="row", Rowv=as.dendrogram(genes.tree), Colv=as.dendrogram(samples.tree))
```

**Exercise 23** *Interpret the heatmap obtained.*

## 7.3 Principal Component Analysis

Back onto the `food` data. We perform a PCA on these data, specifying that the data should be centered and scaled:

```
> food.pc <- prcomp(food, center = T, scale. = T)
```

Let's have a look at the the eigenvalues of the correlation matrix, they represent the amount of explained variance on each component:

```
> food.pc$sdev^2
```

```
[1] 2.197777619 1.144204758 0.848574671 0.807842783 0.001600169
```

And let us compare these values to the scree plot:

```
> plot(food.pc)
```

**Exercise 24** *Choosing the number of components.*

1. *What do you observe between the eigenvalues and the scree plot?*
2. *What does the scree plot represent?*
3. *Where does the "elbow" occur?*

We can project the samples (the food) onto the first two principal components:

```
> plot(food.pc$x[,1], food.pc$x[,2])
> # better is to represent the nutrients with their names:
> plot(food.pc$x[,1], food.pc$x[,2], type = 'n')
> text(food.pc$x[,1], food.pc$x[,2], labels = rownames(food))
```

We can also color the samples according to the clusters obtained with the dendrogram:

```
> # Setting up the colors for the 5 clusters on the plot:
> my.color.vector <- rep("green", times=nrow(food))
> my.color.vector[cut.5==2] <- "blue"
> my.color.vector[cut.5==3] <- "red"
> my.color.vector[cut.5==4] <- "orange"
> my.color.vector[cut.5==5] <- "brown"
> plot(food.pc$x[,1], food.pc$x[,2], type = 'n')
> text(food.pc$x[,1], food.pc$x[,2], labels = rownames(food), col = my.color.vector)
```

And finally we can obtain a biplot:

```
> biplot(food.pc)
```

**Exercise 25** *Sample and biplot plots.*

1. *The percentage of explained variance is defined as the sum of explained variance on the first chosen components divided by the total variance on all possible components. Compute the percentage of explained variance on the first two components.*
2. *Interpret the sample plot.*
3. *Interpret the biplot. What do you observe in common with the sample plot?*

## 7.4 K-means

On the `food` data, let's perform a K-means clustering with k = 5 clusters:

```
> # Note that the stability of the result can be improved by increasing the maximum number
> # of iterations and using multiple random starts:
>
> food.k5 <- kmeans(food.std, centers=5, iter.max=100, nstart=25)
> # food.k5 # to run
```

**Exercise 26** *K-means clustering.*

1. *Interpret the output of the K-means cluster with $k = 5$*
2. *Rerun the same clustering but with $k = 4$, what do you observe?*
3. *For a K-means clustering with k= 4, plot a scatterplot using the function* `pairs()`*. Interpret the clusters and compare with those obtained using the dendrograms.*
4. *Using the PCA analysis performed above, color the samples with respect to the clusters identified by K-means with $k = 4$. What do you observe? How does it compare to the plots obtained above with the colors associated to the dendogram clustering?*

**Part of first Assignment, exercise 4** *Clustering and visualisation.*

1. *Gene selection. We are still interested in the list of the 10 candidate genes from Exercise 2 for the Golub study. Create a new data frame from the* `golub100` *data set so that the data set subset contains only these genes of interest.*
2. *Hirerarchical clustering. Output the heatmap, using the $1-$correlation distance and the average linkage. Comment on the clusters.*
3. *Now, display an heatmap with the Euclidian distance and the Ward linkage. Comment on the differences with the heatmap obtained above.*

   *In the remaining of the exercise, you will transpose the data and check that the number of rows of the data frame is 38 (the number of patients).*
4. *Principal Component Analysis. Apply a PCA on the data frame (use the arguments center and scale).*
   (a) *Will two components be enough to explain most of the variance in the data? (give some numerical figures).*
   (b) *Output the sample plot on the first two components. The samples (patients) should appear on this plot. Comment.*
   (c) *Comment on the biplot obtained. By plotting the boxplots on some chosen genes, explain the characteristics of the genes of interest with respect to how they are located on the biplot: are they overexpressed / underexpressed in some biological conditions?*
   (d) *Compare the clustering of the genes observed on the PCA biplot to the K-means clustering with $k = 2$.*