Practical textbook for the course

Biostatistics

Molecular Biology MSc.

University of Szeged

Written by
Csaba Tölgyesi, Ph.D. & Zsolt Pénzes, Ph.D.

2018.

Contents

Preface

This book is written for Biology BSc and MSc students who are new to data analysis. Our aim was to provide practical directions, while avoided elaborate descriptions of theory. Interested students may find plenty of online and printed sources for further reading, like Crawley's R book series, Dalgaard's Introductory Statistics with R or the freely available R tutorials.

Topics covered in the book include the guidelines all biologists should consider when designing data collection, processing and evaluating data, as well as the basics of preparing visual representations of results. In the first nine chapters we cover only fundamental statistical applications, while in the last three chapters some more advanced techniques are introduced using examples from various sources. For other, more specific data handling and processing methods, specific textbooks and free R package descriptions are available.

All data analysis and representations are done in the freely available software called R. Unlike most statistical programs, R requires users typing in commands instead of clicking on icons or menu items. This may be difficult to get used to in the beginning but after sufficient practice, this will no more be a problem and users can enjoy the advantages of the program, such as its high versatility and the quick repeatability of calculations. At the end of each chapter, we provide a list of all new R functions used in the chapter.

CsT & ZsP

Szeged

August 2018

Chapter 1

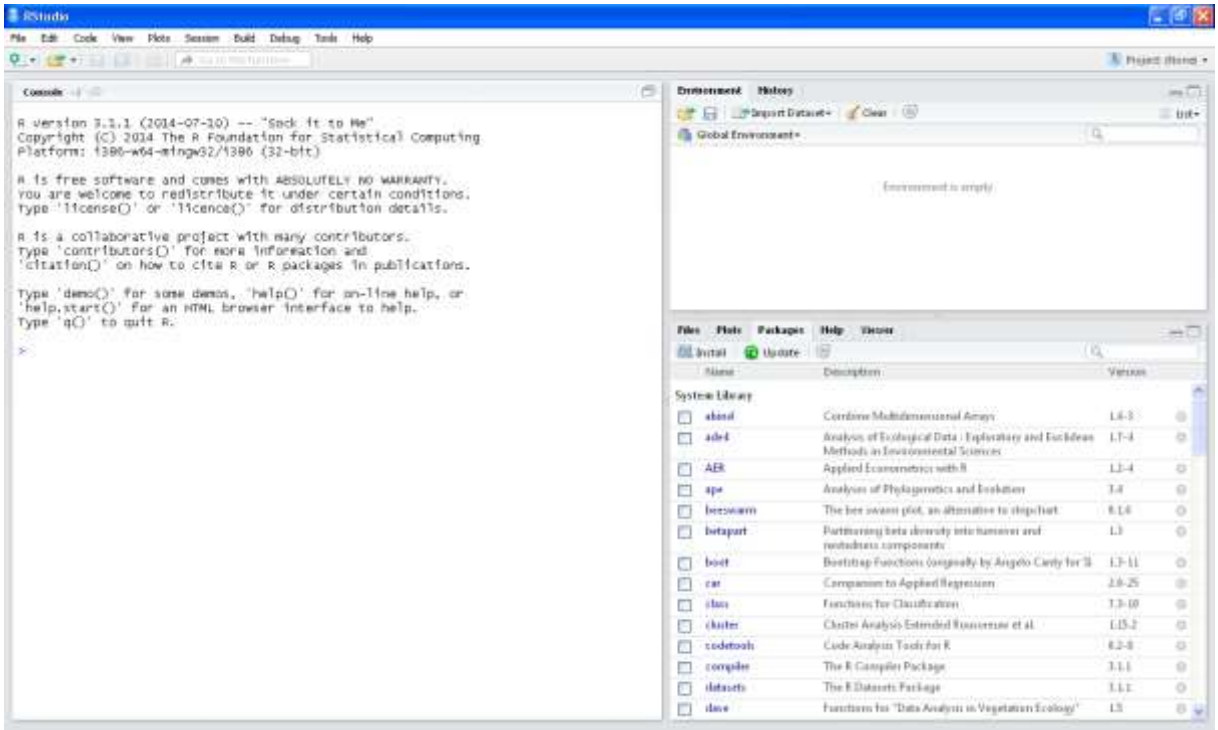
The R language and the R environment

Getting started

R is a freely available software, downloaded from https://www.r-project.org.

For the most user-friendly option, we recommend downloading RStudio as well from https://www.rstudio.com and operating R through it. RStudio will not be functional without having an appropriate version of R on the computer. Select your software versions according to the specifications of your computer and the operation system you use.

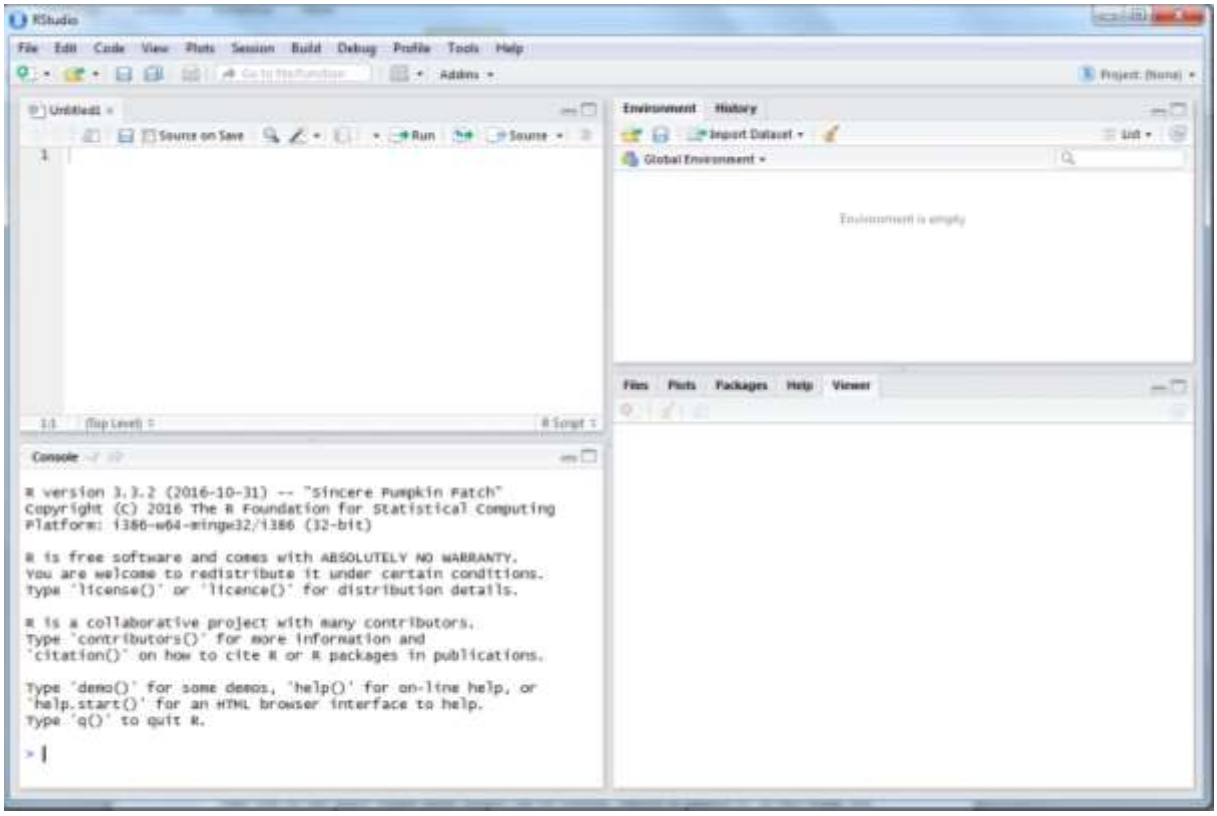
Once installed on your computer, the icons of both R and RStudio will appear among your programs. By clicking on the icon of RStudio, the following window will display:



The main window contains three smaller windows; the left one is identical to the basic R, i.e. the surface offered without using the RStudio. The upper right window will contain a list and some characteristics of data loaded in or prepared locally with R. The lower right window has five menus; Files lists all files available for R; Plots stores all figures and graphs created recently; Packages is a list of available working packages - somewhat like the apps on a smart phone; Help is the help menu, which we recommend consulting as frequently as possible; and using Viewer is a quick way to review stored datasets.

A new working session can be started by clicking on the first icon under the upper menu row and selecting R script. Alternatively, you can click on the hierarchical menus File → New File

→ R script. In either situation, a new window appears on the upper left side, as shown in the following screenshot:



The new window will be the place where most scripts are written, edited or pasted to. In this book, we refer to all command lines typed in this window as 'scripts'.

R can also function as a calculator, so to start-up, try some basic calculations. Type in 2+2 in the upper left window and click Ctrl+Enter. With this, you asked R to calculate 2 plus 2. The result appears in the console (lower left window):

```
> 2+2
[1] 4
>
```

The console prints the script first and then the result in the second line. The > sign at the end indicates that the console is ready for accepting new commands. Whenever this sign is absent, the script was incorrect or incomplete.

Simply pressing Enter is not enough, the command will run only if it is pressed together with Ctrl. Enter alone will insert a line break in the script line, which is useful in complex scripts but will not prompt R to run the command.

Any basic arithmetical calculations can be performed the above way.

One-dimensional datasets

R is of course much more than a calculator. An important basic feature is that it can store datasets and can do the calculations on them without a need for retyping data over and over again. The most basic data type is a series of numbers. This is a one-dimensional data set type, which we will always refer to as a *vector*. For storing vectors, the c() function can be used. Type in `data = c(1,2,3)` and press Ctrl+Enter. Records in a vector are always delimited by comma; using space is optional. (Decimal delimiter in numbers is a dot!)

Instead of the = sign, you can also use <- with the same meaning but in this book we will stay with the equation sign.

Please also notice, that the function (in this case a simple letter c) is followed by a parenthetic part. The function's effect always ends at the closing bracket; R helps to place it correctly as when you open a bracket the ending one also appears. However, in complex scripts you need to double check the appropriate number and location of brackets.

Now you stored this short vector under the name of "data", which appears in the upper right window as a stored item:

```
data  |     num [1:3] 1 2 3
```

So, "data" is a numeric vector (num), which has 3 records ([1:3]) and the records are 1, 2 and 3. If the vector or any other type of item is large, only the first few records will be listed here.

Once you stored an item, you can make operations on it. For instance the script `sqrt(data)` will calculate the square root of all records in the data vector and will return the following output in the console:

```
> sqrt(data)
[1] 1.000000 1.414214 1.732051
```

You can create new vectors by merging already existing ones, too, so not only the raw records can be used with the c() function. By applying the following two scripts, you will end up with a new stored vector (data2) with six records in it.

```
data1=sqrt(data)
data2=c(data,data1)
```

The upper right Environment window will show data2 as follows:

```
data2 |     num [1:6] 1 2 3 1 1.41 ...
```

Data2 is too long for this window to show all records, this is why the "…" ending. If you want to check the records, simply write the name of the item in the upper left window and press Ctrl+Enter. The console will return the full list of records:

```
> data2
[1] 1.000000 2.000000 3.000000 1.000000 1.414214 1.732051
```

Vectors with preset structure can by generated by built-in functions, so it is not always necessary to type in records. Vectors generated by sequencing are frequently useful. For such vectors, you need to define the starting and ending numbers and the increments between each neighboring record. The script seq(from=4,to=9,by=1) will return integer numbers from 4 to 9:

```
> seq(from=4,to=9,by=1)
[1] 4 5 6 7 8 9
```

If you would like to work with a sequence later or if it is simply too large, it may be necessary to store it immediately. The `s1=seq(from=1,to=100000,by=1)` script will lead to a large stored numeric vector in the Environment window as follows:

```
s1    |     large numeric (100000 elements, 781.3 Kb)
```

The basic data of the vector can be retrieved by clicking on the triangular arrow at the beginning of the line.

Please note the structure of the seq() function. It contains three parts within the brackets. These are called the *arguments* of the function (from, to and by). You need to provide values for each of these to run the function properly. It is of course impossible to remember all arguments for all functions. Use the Help menu of the lower right window to check for arguments (or type in `?seq` in the script window and press Ctrl+Enter).

If you type in the function name into the Help menu, the full description will be displayed, along with the arguments. Generally speaking, some arguments are compulsory to provide, others have default values (to be changed only if needed), while the use of the rest is optional. The default order of the arguments is shown in the Help menu, so it is not necessary to type in the name of each argument into the script if you keep the order, so `s2=seq(1,100000,1)` will be identical to the s1 vector. If using the argument names, you can change the order as you wish, so `s3=seq(to=100000,by=1,from=1)` leads to the same vector as s1 or s2.

The sequencing script can be shortened by using colon, if the increment should equal one:

`seq(2:7)` will return

```
> seq(2:7)
[1] 2 3 4 5 6 7
```

Similar to the above type of sequencing, sequences of records can also be generated by repetitions. For example `rep(1,3)` returns

```
> rep(1,3)
[1] 1 1 1
```

The arguments can be stored vectors as well, so using the previously stored *data* vector in `rep(data,2)`, you will receive

```
> rep(data,2)
[1] 1 2 3 1 2 3
```

The data vector was repeated as many times as the second argument required, i.e. two times. It is also possible to repeat each record individually by using the "each" argument:

```
> rep(data,each=2)
[1] 1 1 2 2 3 3
```

(This is the console output, the script line that has to be used is shown in the first line. To reduce redundancy, we will show only console outputs from now on; scripts can be extracted from the first line)

In the rep() function, both arguments can be vectors:

```
> rep(data,data)
[1] 1 2 2 3 3 3
```

The first record in data was repeated once as it was one, the second twice as it was two… Of course, the vectors used do not need to be the same but be careful to have identical records in the vectors, or at least the number of the records should be the multiple of each other. In the latter case, R will recycle the shorter one to match the length of the longer one.

Vectors can contain records other than numbers too. A vector can be a string of characters, which we call a character vector. Each record is indicated by a "" sign when storing it: `names=c("Peter","Tom","Julie")` will be stored in the Environment as

```
names |     chr [1:3] "Peter" "Tom" "Julie"
```

`chr` indicates that this is a character vector. By typing in names into the Script window and pressing Ctrl+Enter, the console will list the content of names (just like for any other stored items):

```
> names
[1] "Peter" "Tom"   "Julie"
```

By now, you have probably noticed that numbers appear blue in the script window, while character items in green. If an item does not match the intended color, it is a clear indication that you made an error. The "" sign is a way to indicate character items in scripts but you can also use the # sign in the Script window to turn entire scripts into character lines. This way the entire line will become green and will be considered as a note or comment and you will not be able to run it as a command. In long scripts, this may be useful for titling and structuring.

A third type of vectors is the logical vector; it can have two types of records: true (T) or false (F). This vector type is not so straightforward as the other two types but can come very handy for sorting from larger numeric or character vectors or for some more abstract applications. We will see some of these in later chapters.

The method of generating and storing logical vectors is identical to those of the other vectors. `logic=c(T,F,F,F,T)` will create the logical vector called logic, which will appear in the Environment window as

```
logic |     logi [1:5] TRUE FALSE FALSE FALSE TRUE
```

Logical operations are very commonly used in R. One situation is when you aim to check records of a vector by relating them to something. If you are interested in or would like to use those records of a numeric vector that meet a some criteria, you can also encounter logical outputs, like here:

```
> data>1.2
[1] FALSE  TRUE  TRUE
```

This script checks the records in the *data* vector, whether they are larger than 1.2.

Regarding the three types of vectors, practical applications may require some crosswalk between them. There are cases when, for ease of data collection, character-type information is numerically stored.
Let's see an example: In a medical study, pain grades are recorded from patients. Pain is difficult to measure as it is very subjective, but one can grade it like "none", "mild", "moderate" "severe". This is typically coded in studies as 0, 1, 2 and 3, respectively. Storing pain grades of five patients in a vector may be done with the `pain_grade=c(0,3,3,2,1)` script. However, R interprets it as a numeric vector, but this is not the case. The numeric nature of the pain_grade vector can be checked by looking at the upper right Environment window, but you can also ask this with the is.numeric() function, which returns the following output in the console:

```
> is.numeric(pain_grade)
[1] TRUE
```

It is possible to get rid of the numeric interpretation of the vector and change it something closer to a character vector. In R terminology, these numbers can be turned into *levels* of pain and pain will be considered as a *factor*. This can be achieved with `pain_grade=as.factor(pain_grade)`. After running this script, you can check again whether the vector is still numeric or not:

```
> is.numeric(pain_grade)
[1] FALSE
> is.factor(pain_grade)
[1] TRUE
```

This can also be checked in the Environment window, which now writes

```
pain_grade  |     Factor w/ 4 levels "0","1","2","3": 1 4 4 3 2
```

Please also note here that R is able to perform circular commands by modifying a stored item and store it under the same name, meaning that it overwrites it without asking for confirmation.

Back to pain_grade vector, it is also possible to provide the exact meaning of the levels using the levels() function: `levels(pain_grade)=c("none","mild","moderate","severe")`
If you call the pain_grade vector again, you get the following output:

```
> pain_grade
[1] none     severe   severe   moderate mild
Levels: none mild moderate severe
```

Now the original pain_grade vector has almost been turned into a character vector, but it is a bit more than that. Levels can be handled by a variety of statistical functions, whereas these

are not available for simple character vectors. The level names also appear in the Environment window.

Two-dimensional data sets

Vectors, as indicated earlier, are one dimensional datasets. R can handle two- or more dimensional datasets too. A two-dimensional dataset is called a *matrix*. A matrix is basically a table of records, typically numbers, arranged into rows and columns.

A matrix can be created from a string of numbers using the matrix() function. As arguments, you have to provide the records to be included, the number of rows (or columns) and arrangements of the records. For example, the `mat1=matrix(1:12,nrow=3,byrow=T)` script creates a matrix from the first 12 positive integer numbers by arranging them into 3 rows. The matrix is filled up with the numbers row by row, as requested by the byrow argument. The matrix is stored in the Environment but can be visualized simply by its name:

```
> mat1
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

If setting the byrow argument to false, the matrix will be filled up column by column:

```
> mat1=matrix(1:12,nrow=3,byrow=F)
> mat1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

It is possible to give names to the rows and columns:

```
> colnames(mat1)=c("A","B","C","D")
> mat1
     A B C  D
[1,] 1 4 7 10
[2,] 2 5 8 11
[3,] 3 6 9 12

> rownames(mat1)=LETTERS[1:3]
> mat1
  A B C  D
A 1 4 7 10
B 2 5 8 11
C 3 6 9 12
```

In the latter script the `LETTERS[1:3]` call provides the first three letters of the English alphabet - `LETTERS` is a built-in character vector of R, containing the English alphabet in upper case. Lower case letters are stored in the `letters` vector.

It is frequently necessary to transpose matrices, meaning that rows have to be turned into columns and vice versa. The t() function can do this:

```
> mat2=t(mat1)
> mat2
```

```
   A  B  C
A  1  2  3
B  4  5  6
C  7  8  9
D 10 11 12
```

Another frequently used operation on matrices is to retrieve certain subsections of it or single records. For this, the "coordinates" of the requested section have to be provided in square brackets. This procedure is called *indexing*. For example, the third record in the second row of `mat2` matrix is retrieved this way:

```
> mat2[2,3]
[1] 6
```

The first number within the square brackets is for the row and the second one is for the column.

It is possible to retrieve entire columns or rows or any subsection of a matrix. If an entire row or column is to be returned, the respective "coordinate" is left out but the comma is needed. For example the B and C columns can be retrieved from mat2 as follows:

```
> mat2[,2:3]
   B  C
A  2  3
B  5  6
C  8  9
D 11 12
```

Since for an entire column all rows are needed, the first "coordinate", which defines rows, is left out. The use of a colon in the second coordinate allows for retrieving both the second and the third columns.

Matrices can be created not only by breaking a vector into a preset number of rows or columns but also by sticking vectors together by either rows or columns. For this, the length of the vectors (number of records in them) must be identical. Vector names will remain as row or column names, depending on the arrangement of the "sticking" procedure. You can use the rbind() or the c(bind) functions for binding according to row or column, respectively. Let's define two vectors and bind them into a matrix:

```
> weight=c(60,72,57,90,95,72)
> height=c(1.75,1.80,1.65,1.90,1.74,1.91)
> mat2=cbind(weight,height)
> mat2
     weight height
[1,]     60   1.75
[2,]     72   1.80
[3,]     57   1.65
[4,]     90   1.90
[5,]     95   1.74
[6,]     72   1.91

> mat3=rbind(weight,height)
> mat3
        [,1] [,2]  [,3] [,4]  [,5]  [,6]
weight 60.00 72.0 57.00 90.0 95.00 72.00
height  1.75  1.8  1.65  1.9  1.74  1.91
```

By now we have accumulated a large number of stored items in the Environment. If one or more are no more needed, it is possible to remove them with the rm() function. For example, mat2 can be removed by running the `rm(mat2)` script.

Data frame is the R term for a simple data table; it is similar in appearance to matrices but its structure is more constrained. Columns are always variables (something that you measure, record, etc), like the weight values of patients, while rows are always study objects, like individual patients, cells, lab rats, etc. This arrangement makes a data frame easy for statistical applications to correctly interpret. A data frame can be created from vectors similarly as matrices but binding is always done by columns, as R assumes that each vector contains values for a variable. Let's create a data frame from the weight values of patients before and after a treatment with the data.frame() function:

```
> before=c(50,56,59,63,67,70,79,88)
> after=c(55,54,61,68,77,78,85,105)
> d=data.frame(before,after)
> d
  before after
1     50    55
2     56    54
3     59    61
4     63    68
5     67    77
6     70    78
7     79    85
8     88   105

> is.matrix(d)
[1] FALSE

> is.data.frame(d)
[1] TRUE
```

As you can see, the `d` data frame looks like a matrix but it is not, as confirmed by checking its identity.

Rows can be named in the same way as shown for matrices:

```
> row.names(d)=c("John","Jack","Tim","Mike","Jason","Julie","Nancy","Sue")
> d
      before after
John      50    55
Jack      56    54
Tim       59    61
Mike      63    68
Jason     67    77
Julie     70    78
Nancy     79    85
Sue       88   105
```

In most real-life applications, you have large data-frames (full lab notes, etc.), but for individual calculations you will need only certain subsets of it. You can specify variables (columns) from data frames using the $ sign. The mean of the before weights and the mean of the weight changes can be calculated the following way:

```
> mean(d$before)
```

```
[1] 66.5

> mean(d$after-d$before)
[1] 6.375
```

Specifying a single record or multiple records or even larger subsets of a data frame can also be done with square brackets like in matrices. So, for example, the after weights of John and Jack can be sorted out in the following two ways:

```
> d[1:2,2]
[1] 55 54

> d[c("John","Jack"),"after"]
[1] 55 54
```

The output is not aligned vertically because it is a vector with two records.

It is frequently needed to have a brief look at the structure of your data frame (e.g. you may be interested if it was loaded in R correctly). For this, you can have a look at the top or the bottom of it using the head() or tail() functions, respectively. These will display the variable names and six objects:

```
> head(d)
      before after
John      50    55
Jack      56    54
Tim       59    61
Mike      63    68
Jason     67    77
Julie     70    78

> tail(d)
      before after
Tim       59    61
Mike      63    68
Jason     67    77
Julie     70    78
Nancy     79    85
Sue       88   105
```

The str() function can also be useful for assessing the correctness of your data frame.

```
> str(d)
'data.frame':  8 obs. of  2 variables:
 $ before: num  50 56 59 63 67 70 79 88
 $ after : num  55 54 61 68 77 78 85 105
```

If you are interested only in the variable names, you can use the names() function:

```
> names(d)
[1] "before" "after"
```

Conditional indexing

It is frequently necessary to sort out records or objects that meet some requirements, such as patients that have a before weight higher than 60. It can happen, that you would like to do calculations only on this subset of the patients. For this kind of sorting, you have to use *conditional indexing* with mathematical operators within the square bracket:

Listing (or storing in new vectors) those after values whose before values are bigger than 60:

```
> after[before>60]
[1]  68  77  78  85 105
```

Listing those before values whose after values are bigger than or equal to 68:

```
> before[after>=68]
[1] 63 67 70 79 88
```

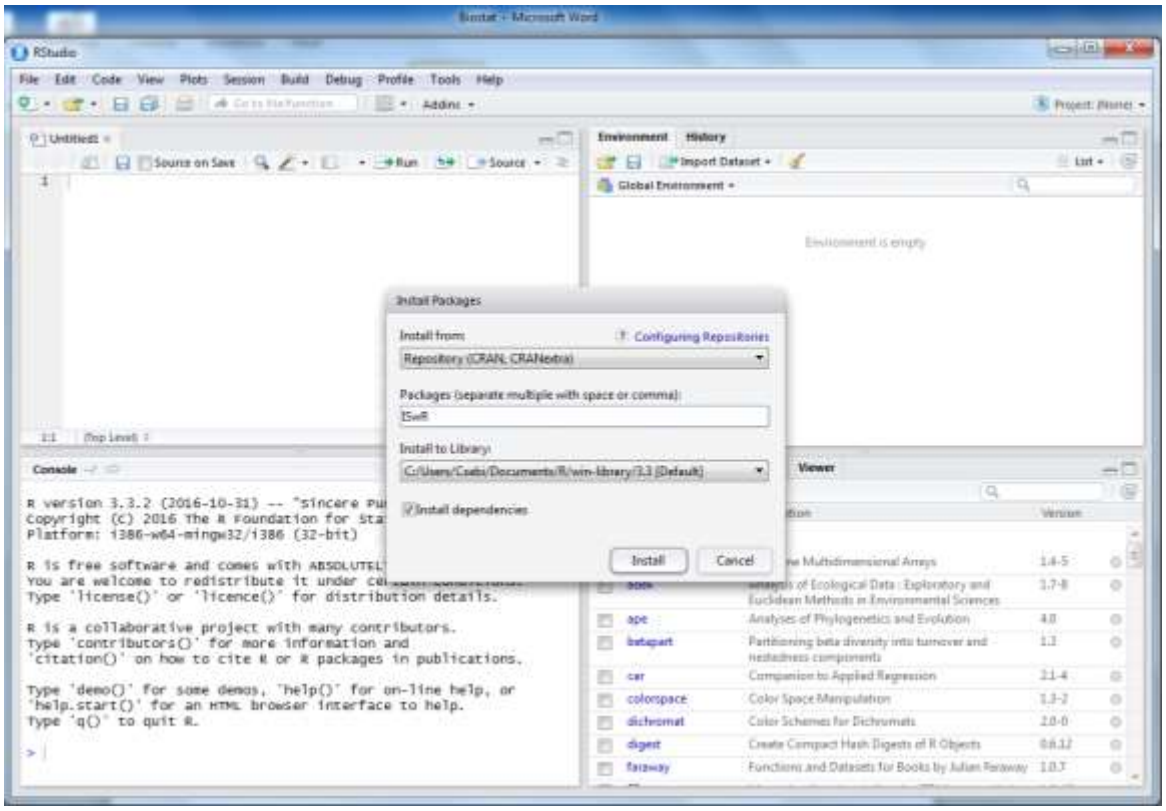
Listing those before values whose after values are equal to 61:

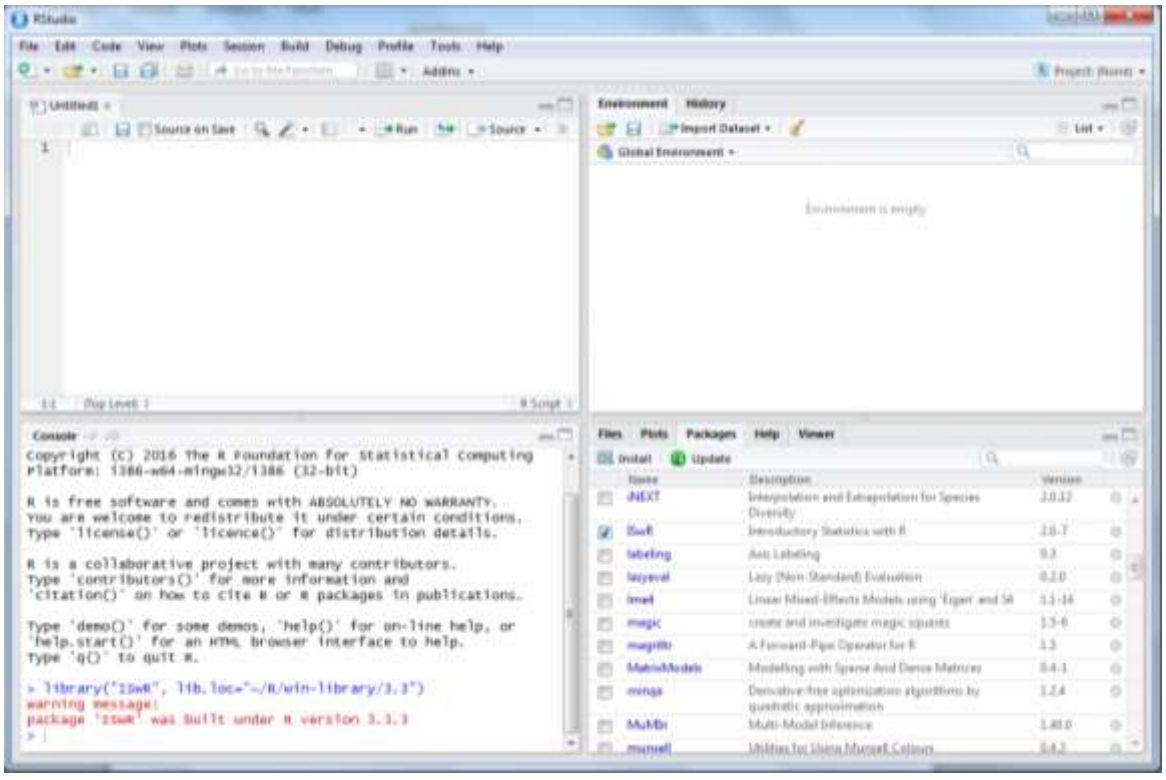
```
> before[after==61]
[1] 59
```

R packages

A main feature of R is its modular structure. Those functions that were used above are built-in functions but more specific functions are contained in separate thematic packages. These have to be downloaded (also freely available) and loaded in the working environment if you need to use them. This feature of R makes it always up-to-date; if a new statistical method is developed by researchers, the first thing is that they prepare an R package and make it available for users. In other softwares, you need to wait for new versions, which may or may not have all new functionalities.

For downloading new packages, go to the Packages menu of the lower right window and click on Install. Type the name of the required package into the empty cell of the pop-up window and install it. Once installed, the package is on your computer but not loaded in the current working environment. To load it in, checkmark it in the User library list. The console will inform you about the successful completion of loading. Sometimes some warning messages appear, but these usually mean no real problem. Try installing and loading in the "ISwR" package.





Note that package names are case sensitive and remember that you need live internet connection. Packages contain functions but also some sample datasets, mostly in the form of data frames. ISwR, for example, contains the `thuesen` data frame. Once you loaded in the ISwR package, you have access to this data frame as well.

```
> head(thuesen)
  blood.glucose short.velocity
1          15.3           1.76
2          10.8           1.34
3           8.1           1.27
4          19.5           1.47
5           7.2           1.27
6           5.3           1.49

> str(thuesen)
'data.frame':  24 obs. of  2 variables:
 $ blood.glucose : num  15.3 10.8 8.1 19.5 7.2 5.3 9.3 11.1 7.5 12.2 ...
 $ short.velocity: num  1.76 1.34 1.27 1.47 1.27 1.49 1.31 1.09 1.18 1.22 .
```

Searching sequences

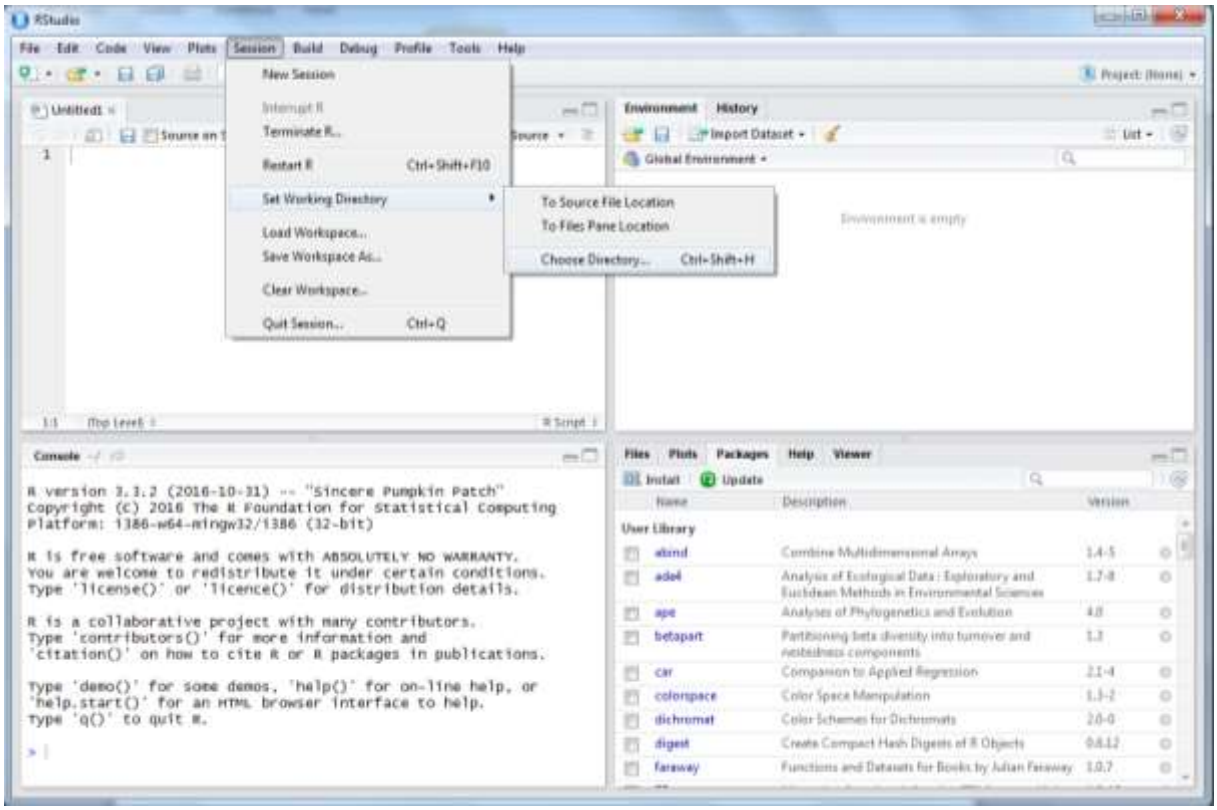
If you want to work with data in `thuesen` or other available data frames in longer scripts, you may not want to use the $ sign for specifying variables, as it can make scripts lengthy. To avoid this, you can attach a data frame to the searching sequence of R using the attach() function. With this, all variables will be accessible without specifying the source data frame. Be careful, however, that variable names can be similar in different data frames, which can make things messy. So use the attach function with caution and detach the dataframe from the searching sequence when finished with working with it:

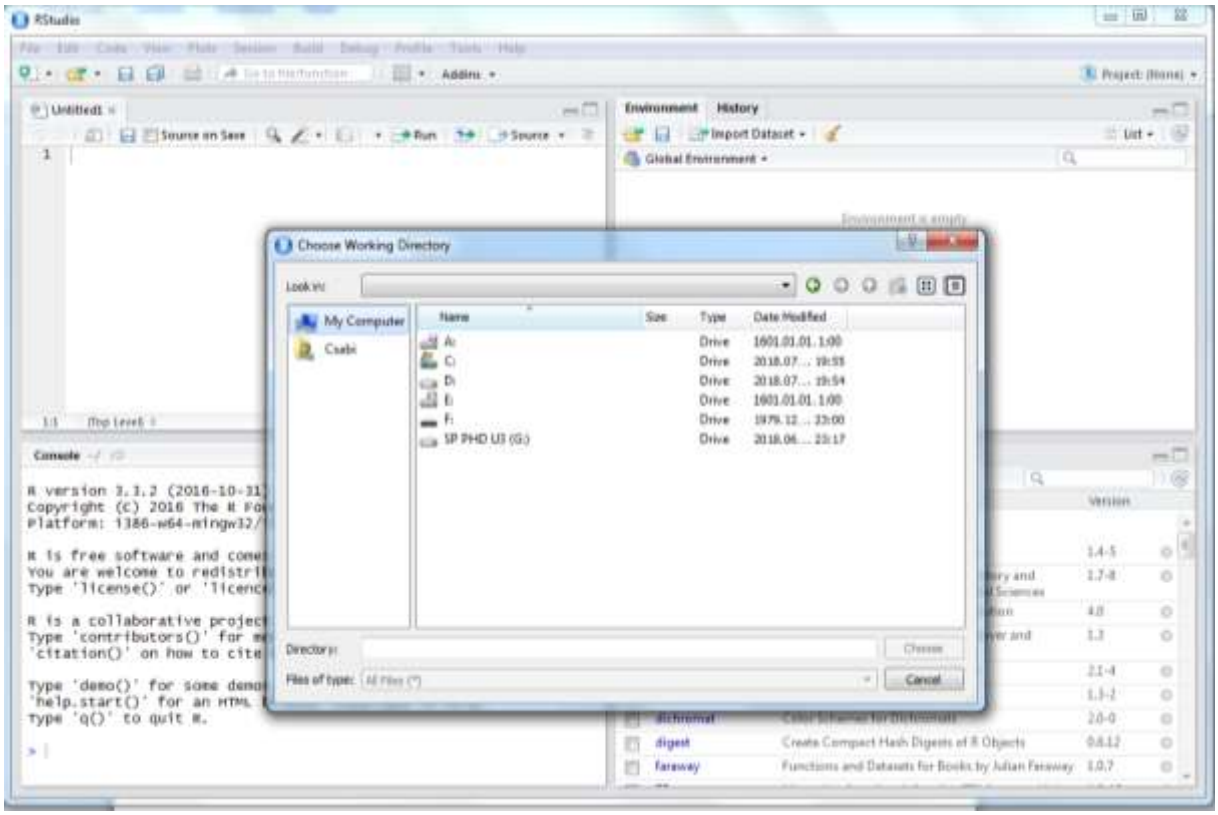
```
> blood.glucose
Error: object 'blood.glucose' not found

> thuesen$blood.glucose
 [1] 15.3 10.8  8.1 19.5  7.2  5.3  9.3 11.1  7.5 12.2  6.7  5.2 19.0 15.1
6.7  8.6  4.2
[18] 10.3 12.5 16.1 13.3  4.9  8.8  9.5

> attach(thuesen)
> search()
 [1] ".GlobalEnv"        "thuesen"           "package:ISwR"      "tools:rst
udio"
 [5] "package:stats"     "package:graphics"  "package:grDevices" "package:u
tils"
 [9] "package:datasets"  "package:methods"   "Autoloads"         "package:b
ase"
> blood.glucose
 [1] 15.3 10.8  8.1 19.5  7.2  5.3  9.3 11.1  7.5 12.2  6.7  5.2 19.0 15.1
6.7  8.6  4.2
[18] 10.3 12.5 16.1 13.3  4.9  8.8  9.5
> detach(thuesen)

> search()
 [1] ".GlobalEnv"        "package:ISwR"      "tools:rstudio"     "package:s
tats"
 [5] "package:graphics"  "package:grDevices" "package:utils"     "package:d
atasets"
 [9] "package:methods"   "Autoloads"         "package:base"
```

First, if you simply ask for the blood.glucose variable (the first column of `thuesen`), R will not know where to look for it; thus you receive an error message. Using the $ sign helps to find it, but if you attach `thuesen`, it will appear in the searching sequence of R after the global environment (this contains those items that appear in the upper right window) and there is no need for the $ anymore. Once detached, `thuesen` disappears from the searching sequence.

Importing data from external sources

Most data used in the R workspace are imported from external sources, e.g. from your lab notes or from the output files of measuring devices. The first step is always to set the working directory of R to the folder your files are located at. Click on Session → Set Working Directory → Choose Directory.





Note that the window used for choosing the directory shows only folders and no data files are shown, so you need to know where your files are!

Files to be imported can be of various formats; the most simple is a tab delimited text file with a .txt extension. The function for importing such a file, like the 'lesson1.txt' file from Coospace (the online educational surface of the University of Szeged), is read.table():

```
> table1=read.table("data2.txt",header=T)
> table1
   names before after
1  Petre     70    66
2   Jill     56    58
3    Sam     90    78
4   Zach     87    80
5   Mike     67    65
6    Ali     81    77
7   Mary     51    50
8  Kerim     69    71
9   Jose     86    86
10  Mark    100    90
```

The header=T argument informs R that the first row in the file is a header, containing the names of the variables (columns).

Another frequently used file type is the .csv (comma-separated values). For these, use the read.csv() function; the names will be interpreted correctly without including the header=T argument. If the .csv file was created in an MS Excel file with German-type (or Hungarian) settings, where the comma is used for the decimal delimiter and values (records) are separated with semi-colon, you need to use the read.csv2() function.

A quick method for data importation is to simply copy it to clipboard (e.g. from an MS Excel sheet) and using the `read.table("clipboard")` script. If not storing under a name, the file will just be pasted in your console but if you store it in the R Environment you will be able to work with it.

Exporting data into external files

Crosswalk between the R workspace and external sources is bidirectional, you can also export data frames or other types of data to external destinations. For this, you have to prepare a script that writes the intended file. The file will be placed to your working directory, so double check whether it is correctly set, otherwise you may encounter some difficulty relocating it. For a tab limited text file, use the write.csv() function.

The `table1` data frame can be exported into a new .txt file called exported_data.txt with the `write.txt(table1,file="exported_data.txt")` script. Files in .csv format can be created the same way using the write.csv() or write.csv2() functions. Be careful to use the version that fits the settings on your computer, otherwise you may encounter problems when opening the file with your spreadsheet processor.

R functions in Chapter 1

c
sqrt
seq
rep
is.numeric
is.facotr
levels
matrix
colnames
rownames
t
cbind
rbind
data.frame
is.matrix
is.data.frame
mean
head
tail
str
names
attach
search
read.table
read.csv
read.csv2
write.table
write.csv
write.csv2

Chapter 2

Data acquisition in biology - what and how?

Biology, like all nature sciences works with data. Data are the representations of some aspects of the real world that we would like to describe in a study. Data are used for analysis, whose results are used to answer study questions, back up hypotheses or reject them. For this, repeatability and a considerable confidence in the results are prerequisites. Therefore, data acquisition need to follow some basic guidelines. Here we discuss only some basics and will introduce the main terms. Data are mostly acquired by measurements/observations, which we carry out on *subjects* (e.g. patients, lab animals, etc.) or *sampling units* (e.g. a preset volume of blood sample, a preset area of a rain forest, etc.). A *sample* is a central term, it means a set of measurements/observations carried out on a set of subjects/sampling units (this can also be the same subject/sampling unit, if we carry out repeated observations/measurements. A sample usually appears as a variable vector in R, such as a column section of a data frame that belongs to a subset of patients forming a group according to the study question. It should be noted, that a sample in statistics differs a bit from the everyday use of the word; a blood sample is not a sample in statistics but a sampling unit, from which we can make measurements.

Another central term is *sample size*. This is the number of measurements/observations in the sample, i.e. the number of records. Again, do not mix it up with the size of the sampling unit! Sample size is usually abbreviated with a lower or upper case letter N and is frequently added to figures because it carries essential information on the reliability of the results. As a rule of thumb, the bigger the sample size, the more reliable the results (but reaching a larger sample size needs more time and money).

The method for sampling can vary but the most frequently recommended and applied approach is the *simple random* sampling (with or without replacement). In this case, subjects/study units are selected from the set of all potential subjects/study units. The complete set of potentially available subjects are called a *population* (not to be mixed up with the usage of the word in ecology!). So, if doing a cancer research, the population includes all patients with that type of cancer ever existed: past, present and any future patients - some of which are technically unavailable for the study, but the results of the study will apply for them too with some confidence. A random sampling from this population means that there will be no bias for other features of the patients, including race, profession, gender, age, place of living, comorbidities, education, etc.

Usually no replacement is applied but in some cases it is not possible to rule out the chance for double measurements. It is easy to avoid measuring the same patient more than once if you know their identity. However, if you measure other living organisms (e.g. fish from a pond) and cannot ID-tag them, it may happen that you pick and measure an individual more than once. This is not a problem if known, as there are statistical methods that can easily handle the situation.

In some cases it may be advantageous to slightly violate complete randomity, if the population is structured. Let's see an example of a human population with a minority making up 5% of the total population. If you chose randomly from the population, it can happen that the minority will be underrepresented or even overrepresented. The latter may particularly be a problem, if there are differences regarding the measured feature between the majority and the minority. Usually, the effect of belonging to an ethnic group is not known (or nowadays may not be politically correct to point it out…); therefore, it can be advisable to rule its effect out at the beginning of the study. One solution is to split up the intended sample size according to the ethnic groups and then make the random selection in each of them. So, if the intended sample size in the above example is 100, five subjects will have to be selected randomly from the minority and 95 from the majority. This is called a *stratified random sampling*.

In some cases random selection may be inappropriate or impractical. If you aim to prepare a geographically explicit map of a biological variable, like a blood iodine map for Germany and you plan to relate it to iodine levels in tap water, you first need to prepare a grid of the study area and make measurements in every grid cell. This is a *systematic sampling*.

A fourth design of sampling is the *nested sampling*. This is never a preferred situation but financial, logistical and ethical constraints may make it necessary. A typical situation is when you have cell clones in Petri dishes but there are more than one clone per dish. The substrate in each dish may be slightly different in composition and texture, they may be exposed to different air currents and temperatures and so on. So, cells from different clones but from the same dish can be more similar to each other (e.g. in growth rate or survival rate) than cells from different dishes. If you had a single clone in every dish, the problem would not occur but this is usually impractical. The non-independence of the data acquired from different clones of the same dish will then need to be taken into account when analyzing the data. Fortunately, there are methods to control for nested design but researchers tend to ignore them…

Once you identified your population, selected the necessary amount of subjects/sampling units, you can start the measurements/observations. There are two main types of data you can collect: *qualitative* and *quantitative* data. Qualitative ones cannot be measured with numbers; typical examples include hair color, blood type, etc. *Binary* data is a special type of qualitative data: there are two levels only (yes/no, dead/alive, male/female, present/absent, etc.). In some cases, data are qualitative, but the values have a certain order. The formerly mentioned levels of pain is a typical example: none, mild, moderate and severe follow this order on the pain scale but they are still qualitative as they cannot be measured using real numbers. Ordered qualitative data are called *ordinal* data, interpreted on ordinal scales. All quantitative data can be stored as character vectors in R but if you want to make statistical calculations on them, most functions will require you to turn them into factors. As discussed earlier, quantitative data can also be coded with numbers, but remember to convert them into factors if using them for calculations.

Qualitative data, on the other hand, are stored as numbers of numeric vectors. Qualitative data can be interpreted on two types of scales, either on an *interval scale* or an *absolute scale.* The interval scale does not have an absolute zero point. A typical example is the Celsius

temperature scale, where the zero point is arbitrarily chosen, therefore out of the four main operations only two, addition and subtraction, can be used. Obviously, multiplication and division do not make sense: 2°C is not half as cold as 4°C, but the temperature difference between 20 and 22°C is the same as between 56 and 58°C. Conversely, the Kelvin temperature scale has a solid absolute zero point, just like the scale of body height, so on these scales, all operations can be performed. Most statistical methods, including the advanced ones, can be carried out on quantitative data (interval and absolute alike), but in some cases you need to know the type of the scale for appropriate interpretation. In contrast, fewer methods can be used on ordinal data and even fewer on nominal data.

Numeric data can be categorized not only based on the scale they are interpreted on but by the possible set of values. Some variables like body height can take any values along the scale; the only limitation is the resolution of the measuring device. These are called *continuous variables*. Other values can take only specific values, such as integer values, along the scale. These are called *discrete variables.* Count data are typically fall into this category. It is highly important to know whether your data are continuous or discrete, because it may affect the choice of statistical methods and may need different parametrization of the calculations.

Distributions

Data usually do not scatter along their scale uniformly. They, of course, can "congregate" around a mean value, but the way how they congregate can depend on the data types. With other words, different values have different probabilities to appear in the sample. The relationship between the values along the scale and their probabilities is described with specific *distributions*. The type of a distribution depends on the inherent nature of the data and the method they were collected.

The distribution of discrete variables can be described with *probability functions* and *cumulative distribution functions*. A probability function assigns probabilities for each value of the scale. Summing up all probabilities will give 1. A cumulative distribution function is somewhat similar but it tells the probability that a record will be smaller or equivalent to a value.

Probability functions cannot be interpreted for continuous variables because there is an uncountable infinite number of potential values, so each single value has a probability of nearly zero (actually zero). Instead of a probability function, we use density functions, which are continuous curves with a surface area under them equaling 1. The probability that a value is smaller than or equal to a value is equivalent to the surface area section left from the value. In contrast, the cumulative distribution function works for continuous variables as well, with the only difference that the function is now a continuous curve and not a series of discrete points.

Both the probability functions and the density functions have the highest y values around the most probable values of the x scale and further away from these, the y values decrease. If there is only one peak of this kind in the functions, we call the distribution *unimodal*. If there are two or more, we call the distribution *bimodal* or *multimodal*, respectively. The latter two

cases are clear indications that the sample is actually made up of two or more groups with distinct properties. If there is one peak, the shape of the curve can be important. Slopes can be symmetrical, but sometimes they are asymmetric with one steeper and one more gentle slope. If the gentle slope extends into smaller values (left side), we call it a *left-skewed distribution*, while if it extends to the right, it is a *right-skewed distribution*. If there is no peak but the probabilities are independent of the scale, then it is called a *uniform distribution.*

Besides the above empirical types of distributions, distributions can be categorized according to the mathematical functions they follow. For discrete data, we discuss the hypergeometric, binomial and the Poisson distributions.

Data collected to answer biological questions like "How many will be/how many times will I get ..something.. out of n occasions?" will follow *hypergeometric distribution* if there is no replacement and *binomial* if there is replacement.

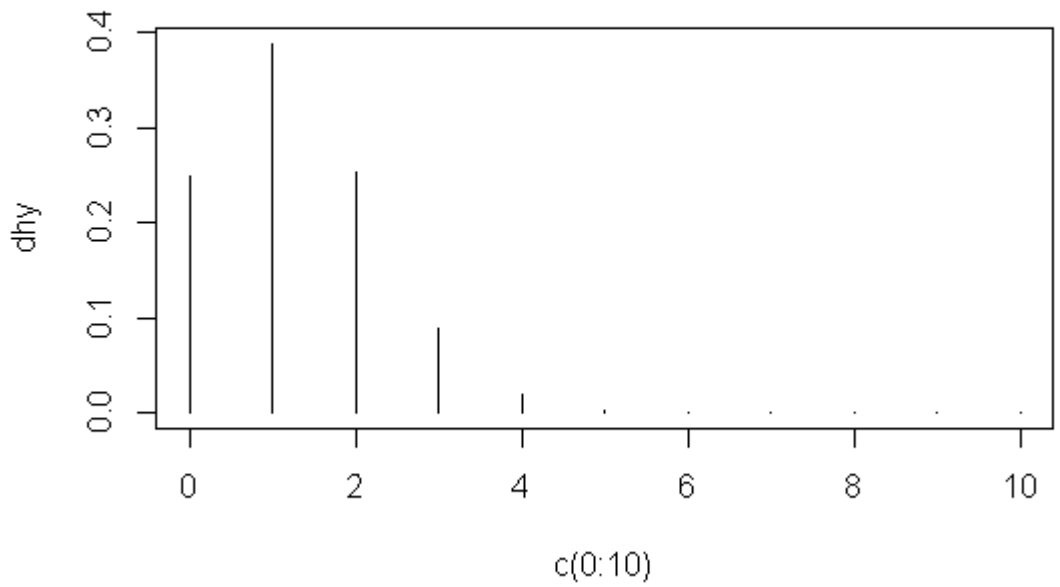
Imagine that you have 120 lab rats, 15 of which are infertile. You pick 10 individuals randomly and ask "How many are infertile among them?" Obviously, you cannot tell without directly checking their fertility but from the prior information you have you can tell the probabilities of having 0, 1, 2, …, 10 infertile ones in your sample. You picked all 10 animals at once, so there is no replacement. Thus, the probability function follows hypergeometric distribution. Hypergeometric distribution is defined with three parameters, the total size of the population, the number of individuals with the character of interest and the sample size. If you have these parameters, the probability for each value can be calculated. R calculates it with the dhyper() function.

```
> dhyper(x=0:10,m=15,n=105,k=10)
 [1] 2.485475e-01 3.883555e-01 2.522309e-01 8.922454e-02 1.892642e-02
 [6] 2.498287e-03 2.061293e-04 1.039307e-05 3.027109e-07 4.527727e-09
[11] 2.587272e-11
```

In the dhyper() function, you first need to provide the values whose probabilities you are interested; these were now all possible outcome values from 0 till 10, then you provide the arguments that specify the distribution. These mostly overlap with the parameters of the hypergeometric function: m is the total number of individuals with the character of interest (infertile), n is the rest of the individuals (total population minus the infertile ones) and k is the sample size. Of course, you can also ask for the probability of a single value; it is not necessary to inquire the probability of all possible outcomes.

It is more informative to plot the probabilities, i.e. to draw the probability function. Easy plotting of basic graphs is a major strength of R. We will go into more detail in a later chapter, so here let it be enough that you need to use the plot() function and provide vectors for the x and y coordinates separated with comma and than to specify the type of the graph, which should be a histogram-like type. The following scripts can generate the plot:

```
dhy=dhyper(x=0:10,m=15,n=105,k=10)
plot(c(0:10),dhy,type="h")
```



As you can see, having one infertile rat in the sample has the highest probability, followed by having either zero or two. The distribution is unimodal and right-skewed. If you omit the type="h" argument from the script, only points will be drawn, not the vertical lines.

If there can be replacement in the sampling (you pick one rat at a time and place it back to the cage, thus having a chance to pick the same one again) or when the sample size is negligible compared to the population size, meaning that the chance for inclusion in the sample does not change much for the remaining members of the population as you progress with the selection, the probabilities follow *binomial distribution*. Now you will need to know only two parameters, the proportion of the character of interest in the population (total numbers are not known/not needed) and the sample size. If you know that the proportion of infertile individuals in a large population of rats is, say, 0.15 and sample 10 animals, you can calculate the probabilities of having 0, 1, … 10 infertile ones as follows:

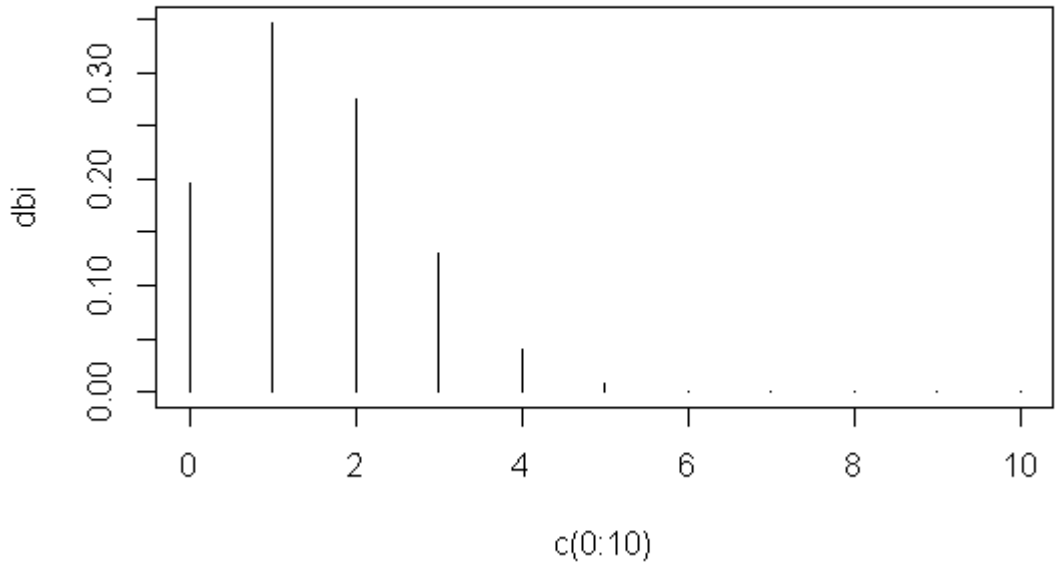
```
> dbinom(x=0:10,prob=0.15,size=10)
 [1] 1.968744e-01 3.474254e-01 2.758967e-01 1.298337e-01 4.009571e-02
 [6] 8.490856e-03 1.248655e-03 1.259148e-04 8.332598e-06 3.267686e-07
[11] 5.766504e-09
```

The arguments differ a bit, as instead of the k argument, sample size is provided with the size argument. You can always check the formulation of the arguments of a function in the Help menu of the lower right window or by directly asking it in the script window by placing a '?' mark before the function like this: `?dbinom`

Plotting the output probabilities is again more informative. The

```
dbi=dbinom(x=0:10,prob=0.12,size=10)
plot(0:10,dbi,type="h")
```

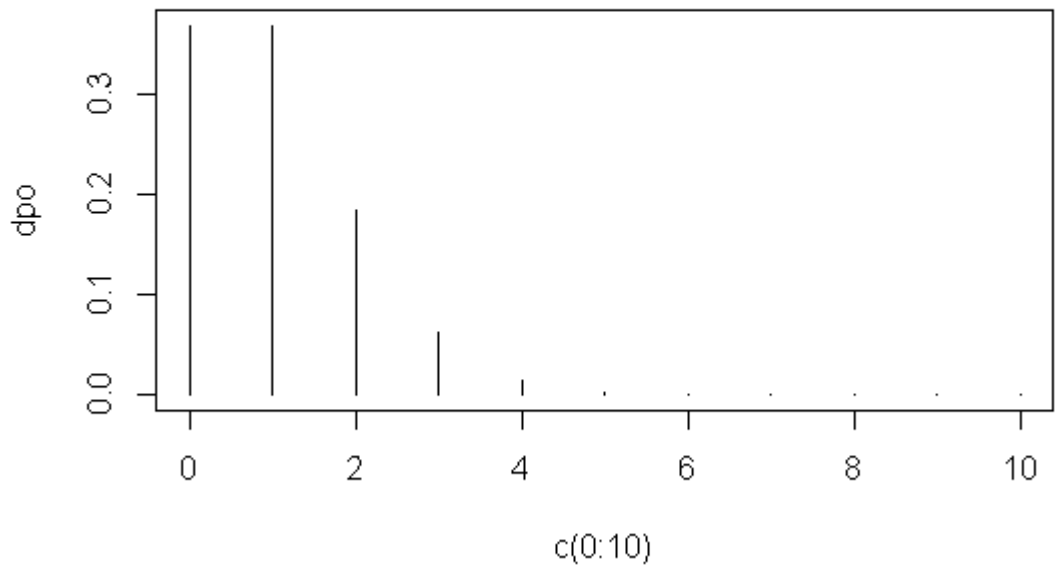
script will return the following plot:



The probability function is similar to the previous one but having two infertile ones in the sample has a higher chance than zero.

A third type of distributions applicable for discrete variables is encountered more frequently in biological applications than the previous two types. This is the *Poisson distribution*. It has only one parameter: sample size is not specified any more but you know only the average value (i.e. the most probable outcome). If you study blood samples and the average number red blood cells is 1 in the high power field, then the probability function of having 0, 1, …. (maximum number is not necessarily defined!) RBCs follows Poisson distribution. You can get the probabilities and draw the probability function with the following scripts:

```
> dpois(x=0:10,lambda=1)
[1] 3.678794e-01 3.678794e-01 1.839397e-01 6.131324e-02 1.532831e-02
[6] 3.065662e-03 5.109437e-04 7.299195e-05 9.123994e-06 1.013777e-06
[11] 1.013777e-07
> dpo=dpois(x=0:10,lambda=1)
> plot(c(0:10),dpo,type="h")
```



According to the probability function, having one or zero cells in the field have equally high chance, which may be surprising, but since the distribution is right-skewed, it is reasonable to have high chance for no cells in the field.

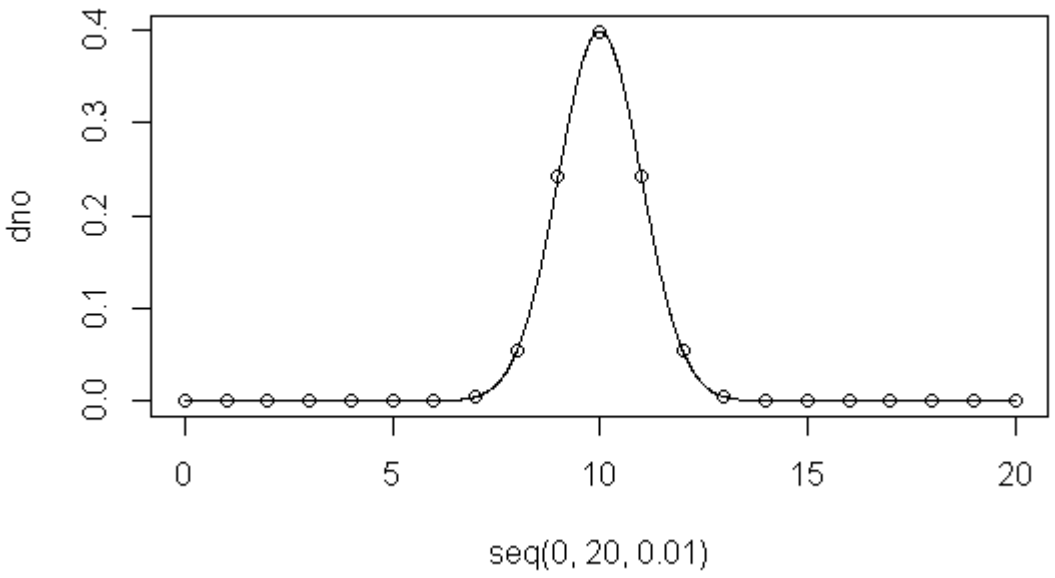
For continuous variables we discuss only one distribution, called *normal distribution* (also known as *Gaussian distribution*). Under ideal conditions most continuous biological variables (body height, blood pressure, amylase activity in saliva, etc.) of populations of organisms follow this distribution and even in non-ideal situations they are close to it and we assume that the distribution does not differ much from normal. There are cases when this assumption has to be declined due severely non-ideal conditions; in such cases it is the responsibility of the researcher to chose statistical methods that do not assume that data follow normal distribution (i.e. distribution-free or non-parametric methods).

The density function of normal distribution is defined with two parameters, the mean (located at the peak of the curve) and standard deviation (the distance between the peak and the inclination points of the curve). The latter is a measure of the spread of the data and will be discussed more thoroughly in the next chapter. Probabilities cannot be calculated the same way as for discrete variables, since now we talk about continuous variables. The function dnorm() returns the probability of the occurrences of values smaller or equal to the provided value. For a normal distribution with a mean of 10 and a standard deviation of 1, these probabilities can be calculated for the first 20 integer values as follows:

```
> dnorm(x=0:20,mean=10,sd=1)
 [1] 7.694599e-23 1.027977e-18 5.052271e-15 9.134720e-12 6.075883e-09
 [6] 1.486720e-06 1.338302e-04 4.431848e-03 5.399097e-02 2.419707e-01
[11] 3.989423e-01 2.419707e-01 5.399097e-02 4.431848e-03 1.338302e-04
[16] 1.486720e-06 6.075883e-09 9.134720e-12 5.052271e-15 1.027977e-18
[21] 7.694599e-23
```

Since normal distribution is a continuous variable, these probabilities will be part of the density function. Plotting the density function together with these points can be done with these scripts:

```
dno=dnorm(x=0:20,mean=10,sd=1)
dno1=dnorm(x=seq(0,20,0.01),mean=10,sd=1)
plot(seq(0,20,0.01),dno1,type="l")
points(0:20,dno)
```



Actually, the curve in this plot is not a real density function but 2000 point probabilities connected with tiny lines, but it looks exactly like the density function in this resolution. Circles are the over-plotted points of the `dno` vector; points therein can be added to the already existing plot using the points() function.

An important variant of the normal distribution is the *standard normal distribution*. It is used in various applications and sometimes data need to be transformed to have this type of distribution. Standard normal distribution is a normal distribution with a mean of 0 and a standard deviation of 1.

SUMMARY

Sampling from population → Sample

Sample size = number of records in the sample

Sampling design: (1) simple random, (2) stratified random, (3) systematic, (4) nested

Data types according to scales:

- Qualitative
  - Nominal (special type: binary)
  - Ordinal
- Quantitative
  - Interval scale
  - Absolute scale

Data types according to possible values:

- Discrete
- Continuous

Distribution types: hypergeometric, binomial, Poisson, normal (Gaussian)

R functions of Chapter 2

dhyper

plot

dbinom

dpois

dnorm

points

Chapter 3

Descriptive statistics and graphics

Datasets are not informative. Usually researchers have loads of numbers in tables, so by simply looking at them, it is difficult to get an idea about the data structure. Descriptive statistics aim to simplify datasets with the use of one or a few more informative numbers or graphs.

Let's create a large dataset first with the rnorm() function

```
> dataset=rnorm(100)
> dataset
  [1]  1.10240865  0.42235592 -0.89690044  0.24762203 -0.63699869 -1.21632528 -1.68049814
  [8]  0.17374434 -0.26306700 -0.74928220 -0.38114325 -0.49483667 -1.15165941 -1.23188123
 [15]  0.44004799 -0.78528288  0.49173230  0.18631365  0.88005746 -0.51816049  0.14187762
 [22] -0.13595818 -1.66765749  0.98479467  0.54834196 -0.22390477 -0.12596422  0.45558192
 [29] -0.55056960  1.61835265  0.47859642  0.17606122  1.64544074 -1.02377046  0.58073909
 [36]  0.70007765  0.39512866 -0.77814508 -0.10366077 -1.27939373 -0.65102818  0.30676266
 [43]  1.11056725 -1.95854180  0.98874347  2.40842759  0.35275148  1.46840945  0.11506030
 [50] -0.52136336 -1.72125606 -0.29611370 -0.24465228 -0.44396001  0.90577748  0.38509456
 [57]  1.15658626 -1.04602447 -0.56635407  0.07800313 -1.44145195  1.86077022 -0.47488823
 [64] -1.14543784  0.61848913  0.30993377  0.19799692 -0.37045973 -1.37024791 -0.28736597
 [71]  0.63990594  0.58474099  1.95697201 -0.94246969  0.06270041  0.24101583  1.81843933
 [78] -0.34365084 -0.86419245 -0.65312785 -2.09805514 -1.67161128 -0.35204459 -0.54846381
 [85]  1.51102912  0.21892089  0.98371907  1.25337709  1.26737644 -0.38780848  0.58132026
 [92] -1.01293365  0.58002238  0.94620384  0.05441268  0.87629644 -1.16511294  0.79069429
 [99] -1.02669169 -0.60021828
```

The function rnorm() gives random numbers that follow standard normal distribution, so now we have a set of 100 such numbers. By simply looking at them will not be too informative. The most simple descriptive statistics include the *mean*, *median* and *mode*; these inform us about the middle values of the sample in some way. The mean (i.e. the arithmetical mean) is calculated by summarizing the values and dividing the sum with the number of values (the sample size). Data can be summed with the sum() function and the length of a vector (i.e. the number of records in it) is extracted with the length() function. The mean can also be calculated simply with the mean() function, which we also used in Chapter 1:

```
> sum(dataset)/length(dataset)
[1] -0.01800791

> mean(dataset)
[1] -0.01800791
```

Although the mean of a standard normal distribution is 0, our mean is slightly smaller. This is because the numbers are randomly generated, which causes some deviation. Increasing the number of records created with the rnorm() function will make the mean approach 0 more and more.

The median is the middle value if the records are arranged in an increasing order. If the number of records is an even number, there will be two middle numbers; in this case the

median is the average of these two records. The median is calculated with the median() function.

```
> median(dataset)
[1] 0.05855654
```

The relationship of the mean and the median depends on the shape of the distribution of the data. If the distribution is symmetric, the mean and the median are close to each other (like in the present case). If the distribution is skewed, they are systematically farther from each other. Since the median depends on the order of the values and not the absolute values, it is less affected by skewness, while the mean is pulled towards the skewed slope of the distribution. This means that in a left-skewed distribution the median is higher than the mean, while in a right-skewed distribution the mean is the higher.

The mode of a dataset is the most common value. This measure is rarely used and does not make much sense for continuous variables. However, for discrete variables, like the grades of students in a school, it can provide some insight into the general performance of the students.

When assessing data, not only the middle value of some sort is important but the spread of the data, i.e. their variability. The most simple measure of the spread is the *range*, which is the largest value minus the smallest value. It is calculated as follows:

```
> max(dataset)-min(dataset)
[1] 4.506483
```

The range() function also exists but it returns the smallest and largest values without doing the subtraction:

```
> range(dataset)
[1] -2.098055  2.408428
```

The average absolute difference between the mean and each value is also informative about the spread, but historically we do not use the absolute difference but its square and these squared differences are then averaged. This average is called *variance* and its square root is the *standard deviation*. If you calculate these from a real sample, the averaging is not done by the total number of records but by the number records minus 1. The reason lies in the fact that the sample variance and standard deviation just approach those of the total population, and statisticians thought this modification will yield better results. Further details can be found in more specialized statistical textbooks.

Variance of the total population: $\frac{\sum_{i=1}^{P}(x_i-\bar{x})^2}{P}$

Standard deviation of the total population: $\sqrt{\frac{\sum_{i=1}^{P}(x_i-\bar{x})^2}{P}}$

Sample variance: $\frac{\sum_{i=1}^{N}(x_i-\bar{x})^2}{N-1}$

Sample standard deviation: $\sqrt{\frac{\Sigma_{i=1}^{N}(x_i-\bar{x})^2}{N-1}}$

P is the size of the population (usually unknown or not countable), N is the sample size, x_i is the ith value of the sample and $\bar{x}$ is the generally accepted abbreviation of the mean (read as 'x bar').

Percentiles (aka. quantiles) go even deeper into the structure of the data. They tell, the location on the scale of the data, below which a certain percent of the data are found. So, the 50 percentile is actually the median. More frequently used are the 25 and 75 percentiles. These are also called *quartiles* because one quarter of the data are smaller than the 25 percentile and 25% are larger than the 75 percentile (75% are smaller). The 25 percentile is the first quartile, while the 75 percentile is the third quartile. The difference between the third and the first quartiles is the *interquartile range*, which, by definition, contains half of the data. Percentiles and the interquartile range can be calculated with the quantile() and IQR() functions, respectively:

```
> quantile(dataset,probs=0.2)
      20%
-0.870734
> IQR(dataset)
[1] 1.233729
```

So, 20% of the data are smaller than -0.87 and the difference between the third and first quartiles is 1.23. This latter measure of the dataset may not seem too informative, but when it comes to visual representation (boxplots), it will be.

Some of the descriptive statistics discussed above can be extracted using the summary() function:

```
> summary(dataset)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-2.09800 -0.65160  0.05856 -0.01801  0.58220  2.40800
```

Listed data include the smallest value, the first quartile, the median, the mean, the third quartile and the largest value.

Calculating descriptive statistics can be done in more structured datasets as well, for instance in data frames, where one variable can serve as categories for another one. If the first 50 records belong to male subjects and the second 50 records to female ones, we can prepare the categorizing vector of `gender=c(rep("M",50),rep("F",50))`. Now you can calculate the mean or all other descriptive statistics of dataset according to genders using the tapply() function. Note that the data vector and the categorizing vector need to be of the same length.

```
> tapply(dataset,gender,mean)
          F           M
-0.02389589 -0.01211992
```

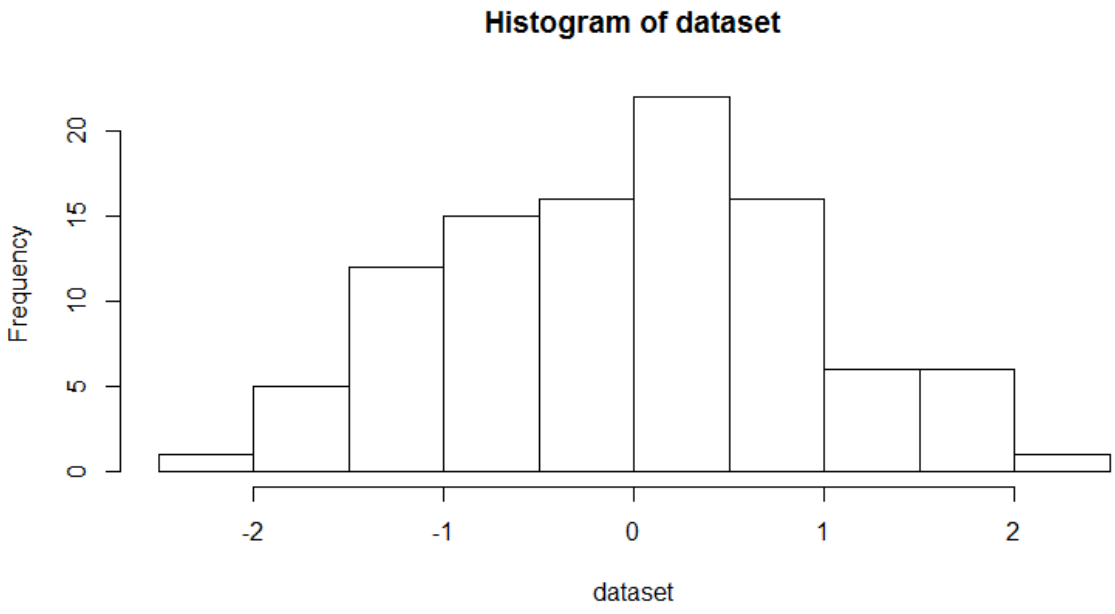
```
> tapply(dataset,gender,IQR)
       F        M
1.274452 1.135969
```

The first argument of the tapply() function is the dataset that needs to be described; it is followed by the categorizing vector and the third argument is the statistics you would like to calculate for each category of the dataset. The mean and the interquartile range are calculated in the above two examples.

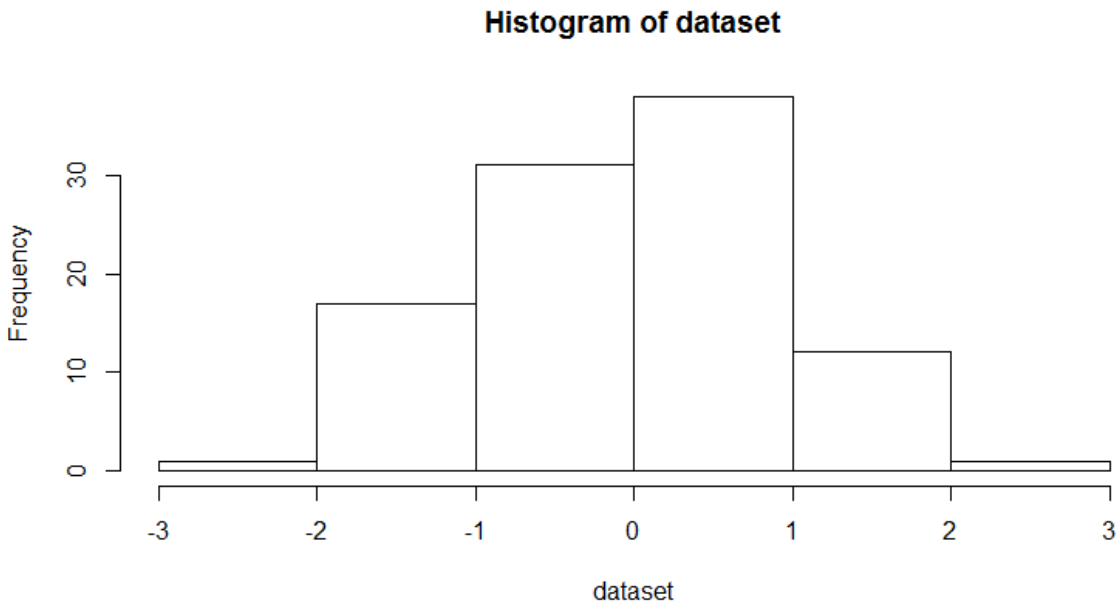
In most real-life applications the dataset and the categorizing variables are columns of the same data frame. In these cases you need to attach the data frame first or you specify the source of the variables using the $ sign as discussed in Chapter 1.

Visual representation of datasets

A simple visualization of data structure is offered by *histograms*. Histograms cut the range of the data into smaller intervals and plots the number of records falling in each interval. `hist(dataset)` returns the following plot:

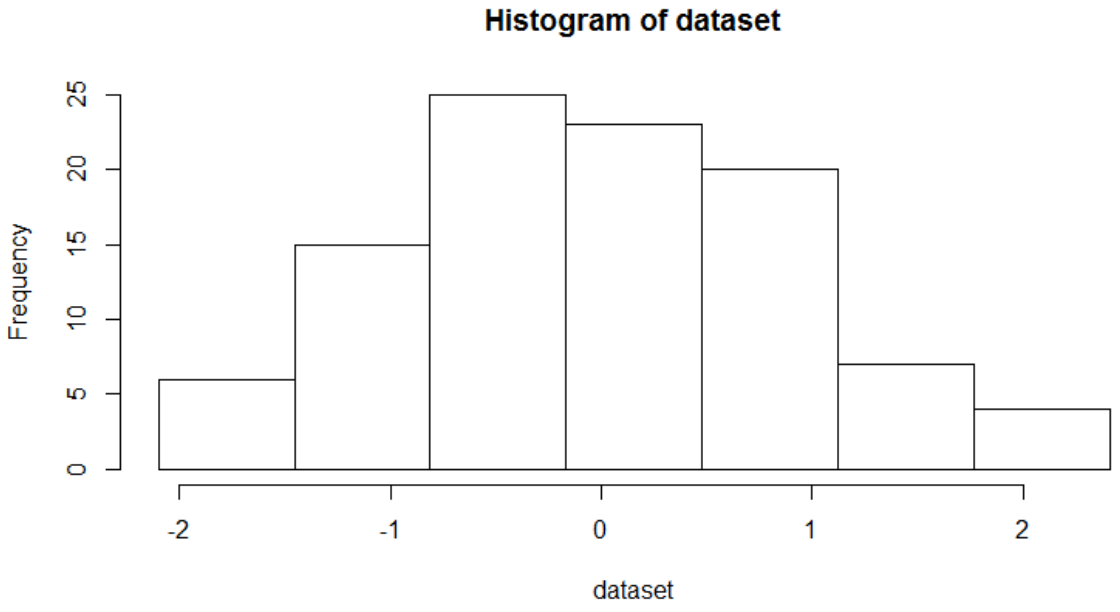


The historgram indicates that the distribution of the dataset vector is not completely symmetrical but this is again caused by the random generation of the records. The number of breakpoints can be customized by specifying the 'breaks' argument. If you want six breaks instead of nine breaks, run the `hist(dataset,breaks=6)` script:



Interestingly, there are only 5 breaks in the plot. R optimizes the number of breaks according to the data, so now it decided for 5 instead of 6. If you insist on a certain number of breaks, it is better to give a vector of exact breakpoint positions and not just the number of breakpoints. Six breakpoints can be forced to R the following way:

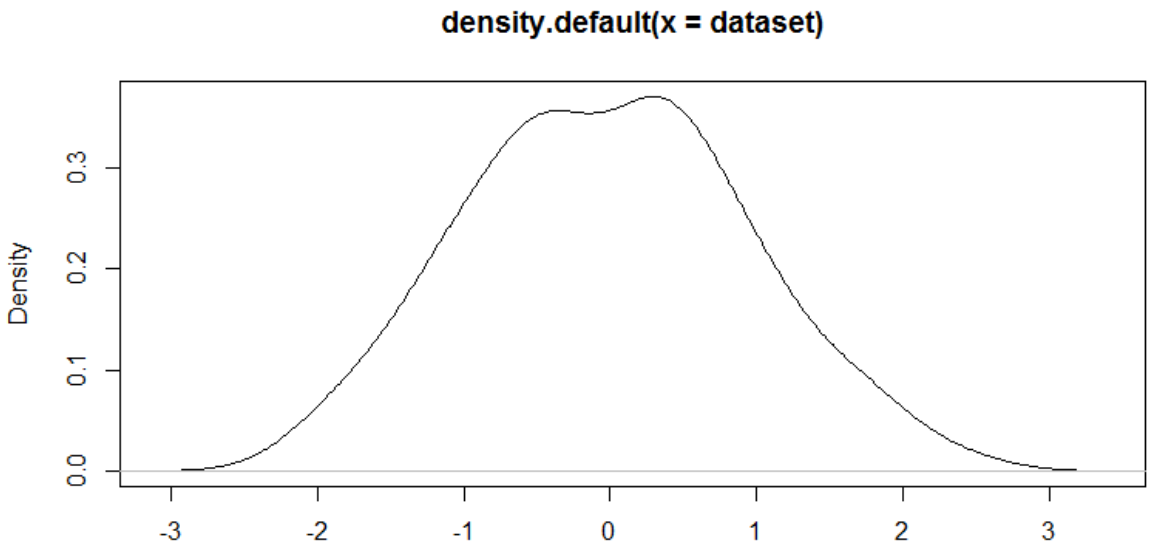
```
> r=(max(dataset)-min(dataset))/7
> min=min(dataset)
> br=c(min,min+r,min+r*2, min+r*3, min+r*4, min+r*5, min+r*6,min+r*7)
> hist(dataset,breaks=br)
```



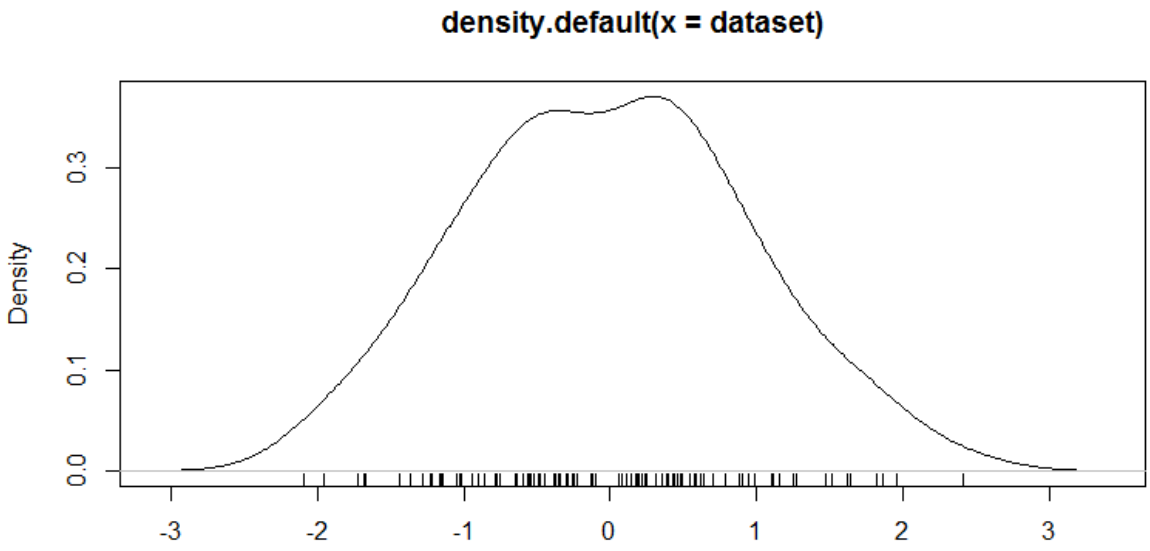
Since six breakpoints lead to seven bars, the range had to be reduced into its one seventh, and this section and its multiples were used to create the vector for the breakpoints. The smallest

and largest values were also included. these are not breakpoints (there is nothing to break there) but R needs these values in the vector. The min variable was stored only for convenience. If you know the exact values where you would like to have the breakpoints, you can also provide a breakpoint vector with raw numbers. The intervals do not need to be identical; if you prefer, you can add uneven breakpoints.

An observant eye can notice some similarity in shape with the density function of the normal distribution. The density function of our dataset can also be drawn, using the density() function with the `plot(density(dataset))` script:

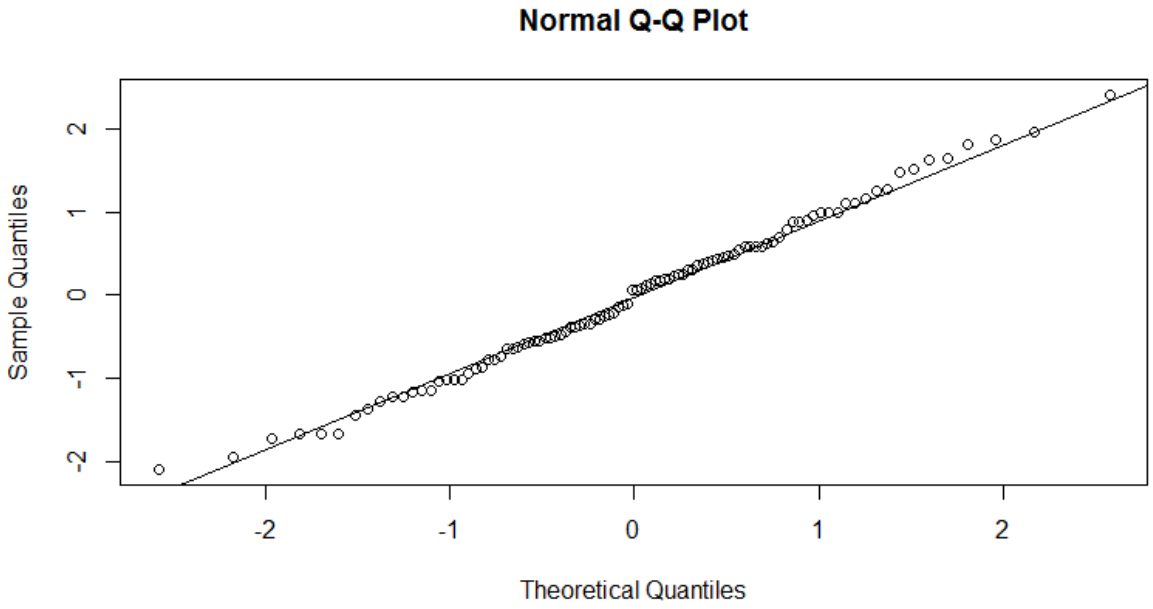


Values in the dataset can also be added to this plot as tiny whiskers along the horizontal axis by running the `rug(dataset)` script after the plotting script:



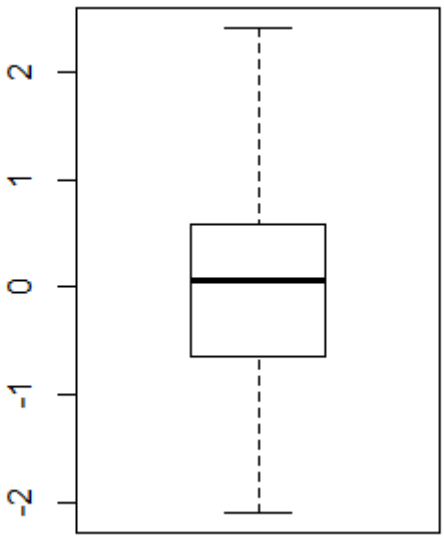
A relatively good fit to normal distribution will be a prerequisite for several statistical applications. This fit can be visually assessed by the so called QQ-plot (quantile-quantile plot), which plots the empirical percentiles against the percentiles of a standard normal

distribution (theoretical quantiles). If they match, meaning that the points are aligned along the y=x line, the dataset follows normal distribution. If there is a severe deviation (particularly if it looks systematic!), like when the upper and/or lower ends gradually slide away from the line or when the points make a clear curve, we can be sure that the data do not follow normal distribution. However, do not be too strict, there is always some deviation from the y=x line! The QQ-plot can be prepared with the `qqnorm(dataset)` script and the x=y line can be superimposed to the plot with the `qqline(dataset)` script:



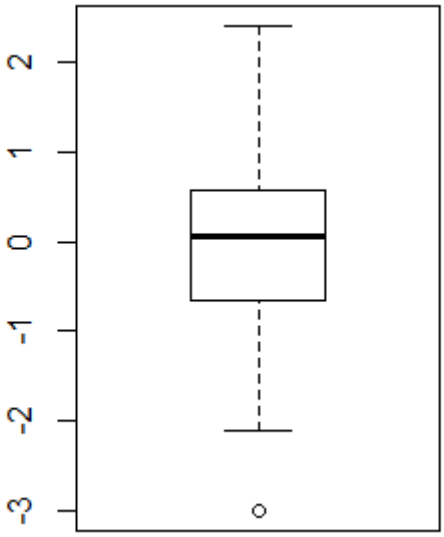
Boxplots

A commonly used visual representation of data is the *boxplot*. A boxplot can easily be prepared for the `dataset` vector using the `boxplot(dataset)` script:



The thick line is the median, the box encompasses the interquartile range and the end of the whiskers indicate the smallest and largest values. However, if these are farther from the box than 1.5-times its size, they appear as outliers and the whisker will terminate at the value, which is still within the 1.5-times distance from the box. If we add such an outlier to the dataset vector, it will change the boxplot as follows:

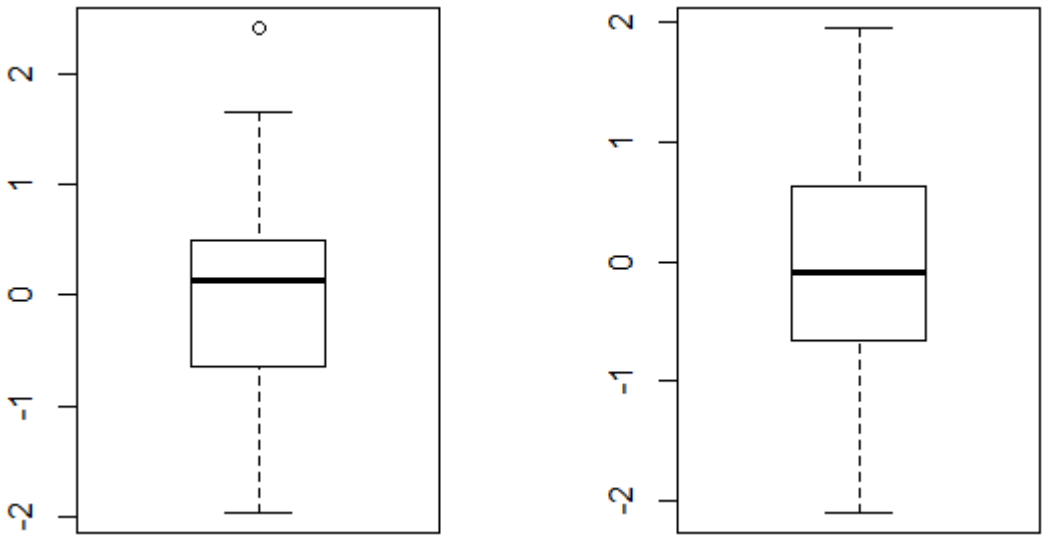
```
dataset2=c(dataset,-3)
boxplot(dataset2)
```



The outlier appears as an empty circle below the lower whisker. Outliers in real-life datasets can originate from measurement errors or incorrect data input. If so, you can disregard them in subsequent analysis. However, outliers can also call attention to interesting biological phenomena. For example, it can happen that certain individuals respond to a treatment in a dramatically different way than the rest of the subjects (leading to outliers). This deserves some attention and should not simply be ignored.

A single boxplot is rarely published; boxplots are usually used to visualize the differences between two or more datasets. Let's split the dataset vector into two subsets containing 50 records each and prepare two separate boxplots from them using the following script:

```
par(mfrow=c(1,2))
boxplot(dataset[1:50])
boxplot(dataset[51:100])
par(mfrow=c(1,1))
```

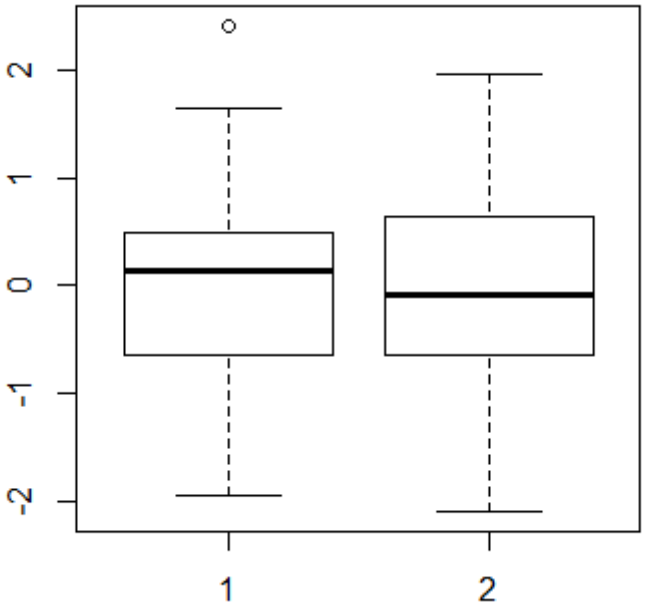


This will place the two boxplots next each other. The par() function is used to modify the lower right window; setting the mfrow (multiframe according to rows) argument splits up the window into a grid according to the values, which are c(1,2) in this case, meaning 1 row and 2 columns. This virtually split window will than be filled up with plots according to rows (although there is only one row in this case). At the end, it is highly advisable to set it back to the unsplit layout with the par(mfrow=c(1,1)) script, otherwise the layout will remain the same for future plots as well.

You can see that the two boxes look rather similar, but this is no surprise, given the origin of the data. Notice also that the first 50 values contain an outlier but this is only a result of the random generation of the data.

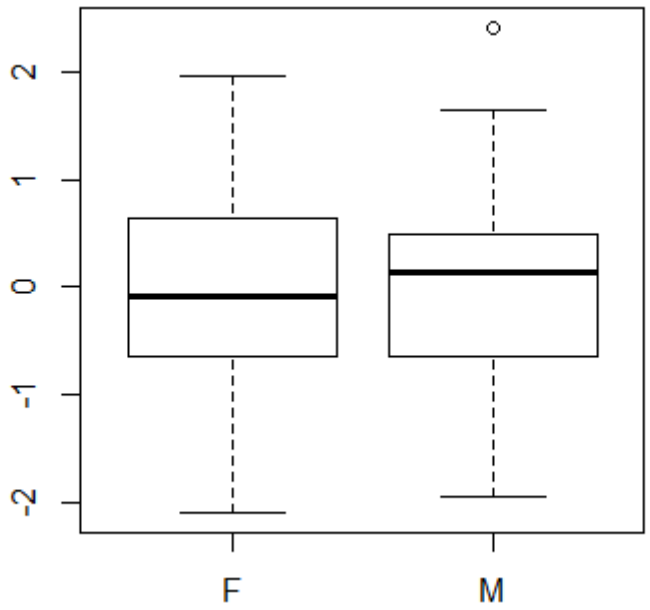
The two boxes can also be placed within one single frame using the following script:

```
boxplot(dataset[1:50],dataset[51:100])
```



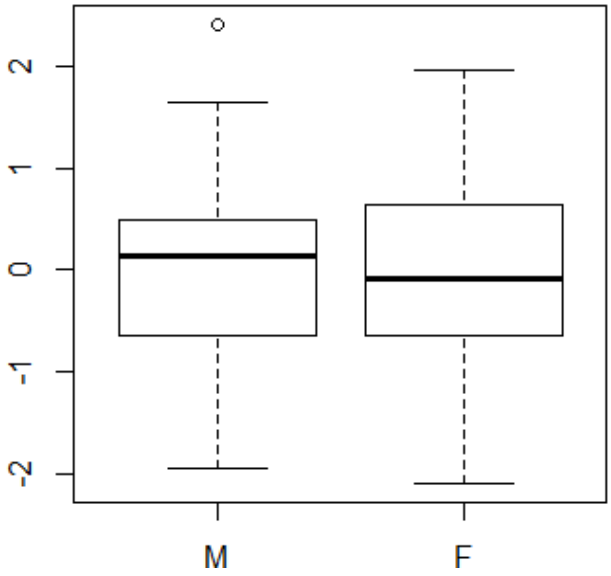
Here we just split up the original 100 data by indexing, but in most real-life cases the dataset is structured. For example, some of the data can belong to male patients and some to female ones as shown for the tapply() function. If so, this structure can be used for boxplotting females and males separately using the `boxplot(dataset~gender)` script.

The parenthetic part with the ~ sign is read as 'dataset as a function of gender'. Such a script structure is called a *formula* in R and generates this plot:



Now the abbreviations used for identifying genders in the Male and Female vectors appear below the boxes. Although the gender vector starts with 50 "M" characters, the first box is for females ("F"). This is because R always arranges boxes (or the units of other similar graphs) so that their tags follow alphabetical order. If this is not appropriate, you can reset the order of the levels using the relevel() function. This function works only on factors, so the gender vector, which is a character vector at this point, has to be transformed into factor:

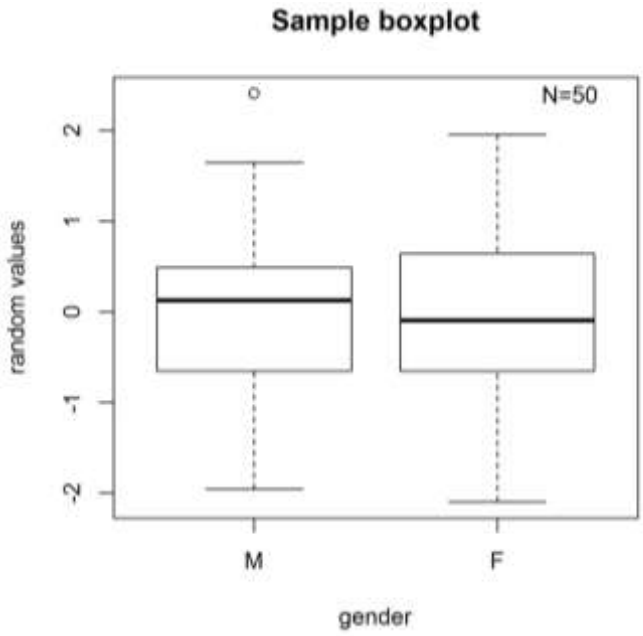
```
gender1=factor(gender)
gender1=relevel(gender1,ref="M")
boxplot(dataset~gender1)
```



This boxplot is identical to the previous one, the only difference being the order of the boxes.

Boxplots are frequently used in publications, so a bit more elaboration on formatting may be necessary. A title can be added using the 'main' argument, x and y axis labels are added using the 'xlab' and 'ylab' arguments. Text is added using the text() function. The figure can be exported using the icons of the lower right window but it is mostly more advisable to write the figure directly into an external file, typically a .tiff file, as most scientific journals prefer submitting tagged image file formats.

```
tiff(file="gender_plot.tiff",width=1500,height=1500,res=300)
boxplot(dataset~gender1,main="Sample boxplot",ylab="random values",xlab="gender")
text(2.3,2.4,"N=50")
dev.off()
```



First, an empty tiff file is created in the working directory with specified dimensions. The width and height arguments define the dimensions in pixels and the res argument gives the resolution in dpi (dot per inch). After this, you fill up the file with the plot and you can add extra text or lines or whatever you want. The position of these items is defined with

coordinates. The horizontal axis is not numeric, but for the placement of extra items it is considered as numeric, with integer values at the boxes starting with 1. Once you finished with the composite figure, you need to close the file by terminating its processing with the dev.off() function.

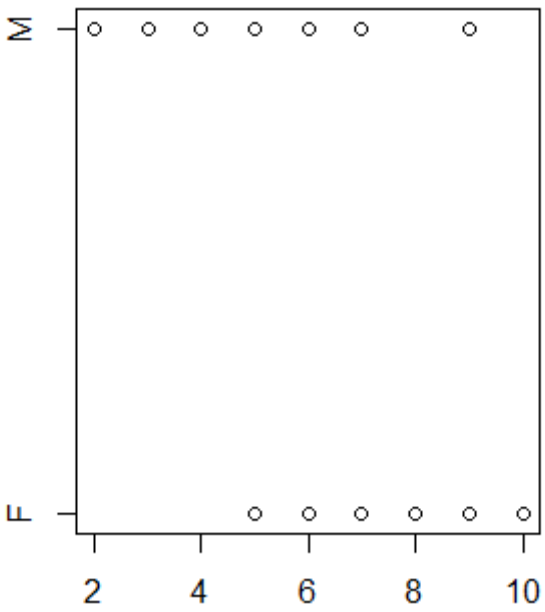
With some further work, it is possible set colors, box widths, line widths, font types, font sizes and so on.

Preparing long and complex scripts like this may be tiresome but once you have it, you can save it, say, in a text file and next time the only thing you need to do is to change the vector names in the boxplot formula and maybe adjust the argument values a bit and you will have your new plot.

Stripcharts

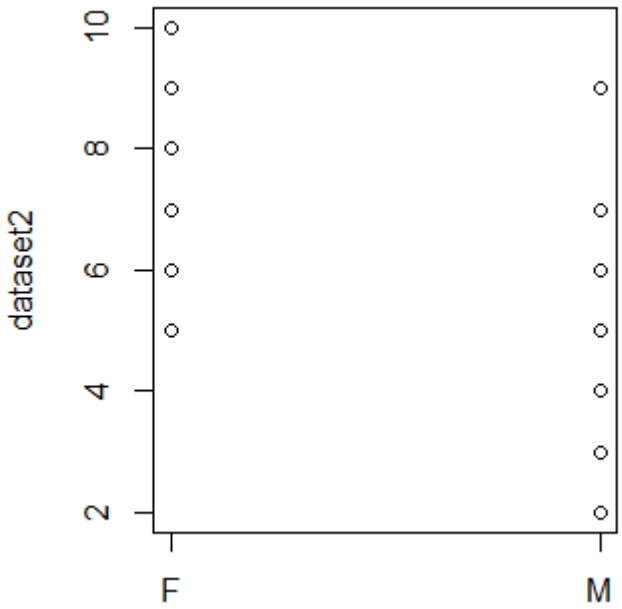
Boxplots are not the only graphical illustrations of data. It is possible to plot raw data as well; this is more advisable when the number of data is rather low and the percentiles are thus less informative. Let's create a new data set containing only 20 records, and structure them similarly as before using genders and plot them accordingly:

```
> dataset2=c(2,3,4,3,5,4,6,5,7,9,5,6,7,6,7,9,9,8,9,10)
> gender2=c(rep("M",10),rep("F",10))
> stripchart(dataset2~gender2,pch=1)
```



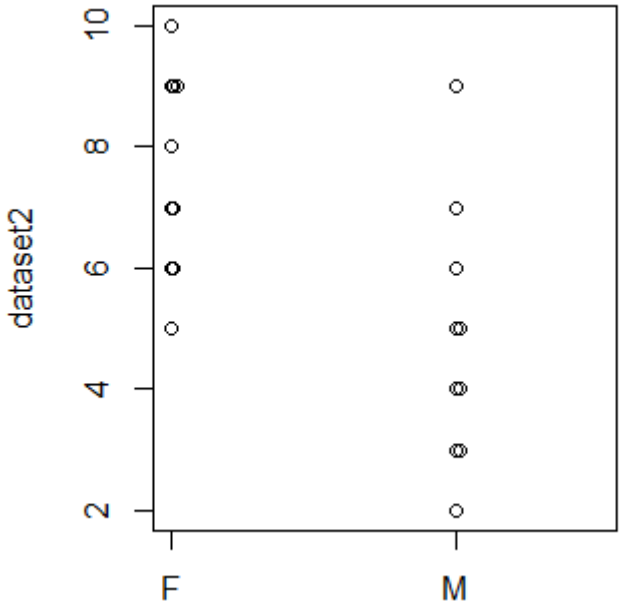
The pch argument defines the symbol used for the points. By checking the pch argument in the Help menu, you will see that there are built-in symbols coded by numbers from 0 till 25; 1 is for empty circles. An unfortunate default setting of strip charts is that they show data horizontally. The layout can be changed to vertical simply by specifying the direction:

```
> stripchart(dataset2~gender2,pch=1,vertical=T)
```

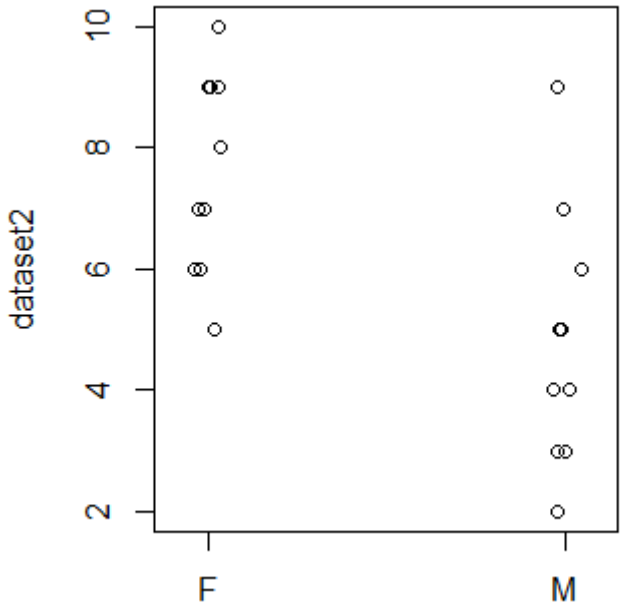


An observant eye can notice that the plot contains only five points for females and seven points for males but there are 10 records for each of them. This is because there were identical values in the vector sections, which are fully over-plotted in the plot. However, it may be advisable to make all points visible. This can be achieved by adding some random horizontal component to the position of the points with the method argument. If method is set to "stack" only the over-plotted points will be shifted a bit, while jitter adds a random vertical component to all points; the extent of jittering can be set with the jitter argument:

```
stripchart(dataset2~gender2,pch=1,vertical=T,method="stack")
```



```
stripchart(dataset2~gender2,pch=1,vertical=T,method="jitter",jitter=0.05)
```



SUMMARY

Descriptive statistics:

- Middle values
  - Mean
  - Median
  - Modus
- Spread
  - Range
  - Variance
  - Standard deviation
- Distribution-related
  - Percentiles (quantiles)
  - Quartiles
  - Interquartile range
  - Outliers

Visual inspection of data

- Histogram
- QQ-plot
- Boxplot
- Stripchart

R functions of Chapter 3

rnorm
sum
length
median
max
min
range
quantile
IQR
summary
tapply
hist
density
rug
qqnorm
qqline
boxplot
relevel
par
tiff
text
dev.off
stripchart

Chapter 4

One- and two-sample tests

In biological studies, we make observations/measurements on samples to estimate properties of the total populations, like their true means. We are frequently interested whether a true mean is different from a certain value (called a *hypothetical value*) or from the true mean of another population. However, we do not exactly know these true means (or any other true parameters of the population); they are only estimated with our sample. And indeed, these empirical means do differ a bit from the true means because of random chance and therefore a difference between the empirical mean and a hypothetical value or the empirical mean of another population may not mean real difference. There are methods to test, whether these differences are real or just caused by randomity. To be more precise, these methods give a probability for both of these and if the probability of a real difference is big enough, we accept it. These methods are called *statistical tests* or *hypothesis tests*.

The logic of these tests is rather simple, although their mathematical background can be complex. We formulate two hypotheses and we have to choose one. The *null hypothesis* (abbreviated as H0) says the mean (or some other parameter) of the population does not differ from a certain value or from the mean (or some other parameter) of another population, i.e. any difference is only due to random chance. The *alternative hypothesis* (H1) says that there is real difference. H0 is rejected, if its probability is "too low". This probability is checked by calculating an appropriate *test statistic* from the sample. A test statistic is a numeric value and has a typical standardized distribution. A standardized distribution in general means that its most probable value is set to 0, like in the case of the standard normal distribution. The larger the test statistic in absolute value, the smaller its probability, meaning that the probability of H0 is also small. This probability is easily calculated with R, and is called a *p-value*. They equal the probability that rejecting H0 and accepting H1 is incorrect. However, if the chance for this incorrect choice is low enough, historically 0.05 (5%), then we vote for H1 and say that the difference is *statistically significant*. Since R provides exact p-values, it is easy to check whether the difference is significant. The values of the test statistic and the p-values are to be reported in scientific communication (publications, presentations, posters, etc.).

Knowing all details of the calculation of the test-statistic and the p-values is not essential for biologists (although it is advisable to have some basic understanding), but the selection of the right test and its correct parameterization are prerequisites for getting reliable data.

One-sample tests

One-sample tests are used when a sample from a population is available and you would like to compare a parameter of it to a hypothetical, preset value. The most common version is when you are interested, whether the mean of your study population differs from a certain value.

Let's create a vector containing weight values of patients and check, whether the true mean of the population they come from differ from 72 kg:

```
> weight=c(52,54,56,61,63,65,68,75,75,82,87)
> mean(weight)
[1] 67.09091
> t.test(weight,mu=72)

        One Sample t-test

data:  weight
t = -1.4145, df = 10, p-value = 0.1876
alternative hypothesis: true mean is not equal to 72
95 percent confidence interval:
 59.35807 74.82375
sample estimates:
mean of x
 67.09091
```

The mean of the sample is approx. 67 kg, so there is a 5 kg difference from the hypothetical value. This difference is tested for significance with a *one-sample t-test*, using the t.test() function. Its first argumentum is the dataset to be tested, which is the `weight` vector, and this is followed by the mu argument, which is the hypothetical value, now set to 72. The output contains a t-value, which is -1.4145. This is the test statistic mentioned above. The negative sign (smaller than 0) means that the sample mean is smaller than the hypothetical value. The df is the degree of freedom; in this case it is the sample size minus 1 and determines properties of the distribution of the t statistic. Degree of freedom is also a central term in statistics but here we do not discuss it in more detail. The p-value is of great importance; it is now 0.1876, which means that if we decide for H1 (the 5kg weight difference is real), the chance for being wrong is 18.76%. As mentioned earlier, the threshold is 5% (p=0.05), so this is too much. We stay with the H0 (the difference is caused only by random chance).

The *95 percent confidence interval* is also essential in statistics. This interval is provided by the t.test() function; it means the interval that the true mean of the population (from which the sample was selected) is located in with 95% probability. If the sample mean is outside this interval, it is also an indication of significant difference. In such a case the p-value is also smaller than 0.05. Let's check whether this really works by setting the mu argument a bit outside the confidence interval:

```
> t.test(weight,mu=75)

        One Sample t-test

data:  weight
t = -2.2789, df = 10, p-value = 0.04587
alternative hypothesis: true mean is not equal to 75
95 percent confidence interval:
 59.35807 74.82375
```

```
sample estimates:
mean of x
 67.09091
```

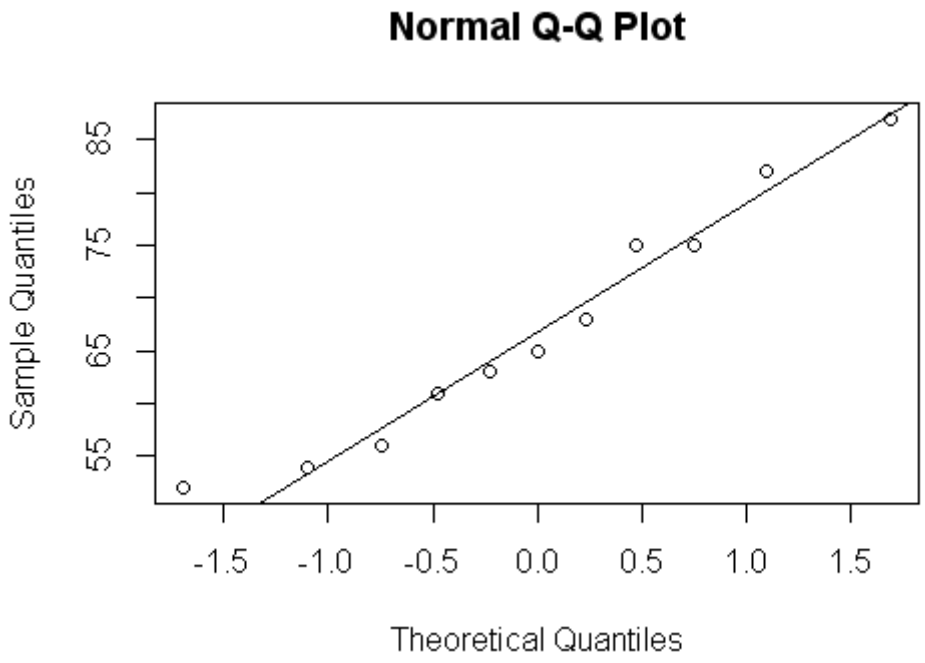
As you can see, 75 is a bit higher than the upper confidence limit, and accordingly, the p-value is now a bit smaller than 0.05, so we have a significant result. In a publication, for example, we can report this as follows:

"The weight of the studied population differed significantly from 75 kg (t=-2.28, p=0.046)."

The test statistic and the p-values are used to back-up the statement and, of course, there is no need to report all decimals shown in the R output.

However, there is one more thing to consider. Statistical tests give reliable results only if the sample fulfills some conditions, which we call *assumptions*. A one-sample t-test has at least two assumptions that have to be met: (1) The sample has to be selected randomly (well, we thrive for this in most study designs, so this is usually not violated); (2) and the records in the sample need to follow normal distribution (or at least they should not deviate from it much). Thus, the normality of the sample needs to be checked in all cases. This is done with the QQ-plot:

```
qqnorm(weight)
qqline(weight)
```



The points fit the qqline rather well, meaning that the theoretical and sample quantiles match and thus the normality assumption is not violated.

However, if there is severe deviation from the theoretical quantiles, the t-test will not give reliable results. In such a case an alternative type of test can be recommended, which is not dependent on the normality of the sample. These are the *non-paramteric tests*. They are based only on the ranks of the records and not their absolute values. As a rule of thumb, non-parametric tests give reliable results, if the sample size is large, so if you have a small sample

size with non-normal distribution, it is better to collect more data than blindly proceed to non-parametric tests and believe what they return.

The non-parametric alternative of the one-sample t-test is the *Wilcoxon signed-rank test*. It is performed in R similar to the t-test. Although the t-test is also suitable for the `weight` vector, let's do a Wilcoxon test on it:

```
> wilcox.test(weight,mu=75)

        Wilcoxon signed rank test with continuity correction

data:  weight
V = 6, p-value = 0.05758
alternative hypothesis: true location is not equal to 75

Warning messages:
1: In wilcox.test.default(weight, mu = 75) :
  cannot compute exact p-value with ties
2: In wilcox.test.default(weight, mu = 75) :
  cannot compute exact p-value with zeroes
```

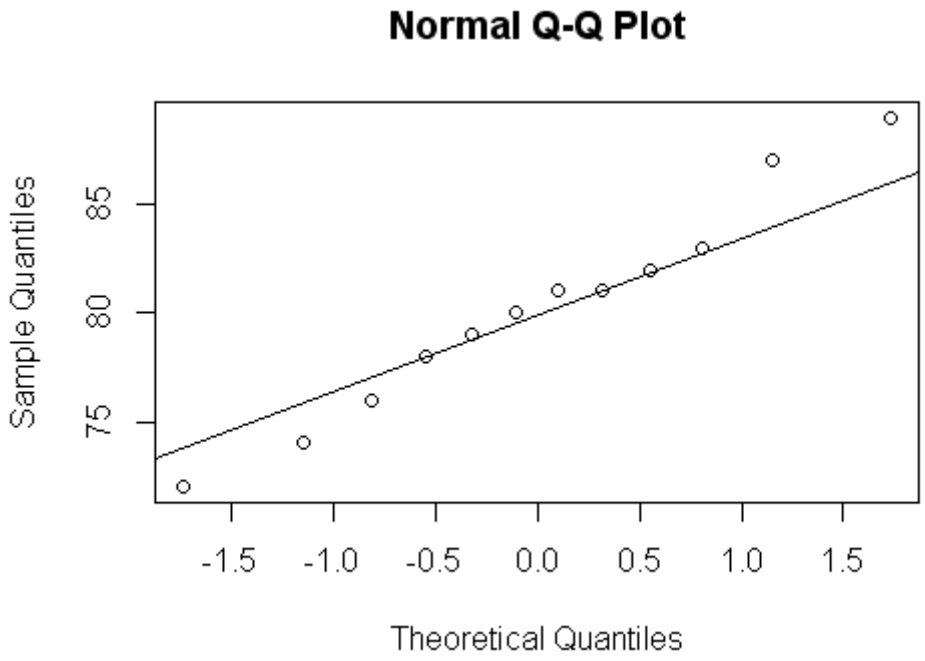
The test statistic of the Wilcoxon test is the V-value, which, like the t-value, needs to be reported along with the p-value. Although the t-test yielded significant result, here we got a non-significant one. Maybe it would have been significant with a lager sample size but with N=11, the test was not very powerful.

There are two warning messages at the end of the output but these are not serious problems; the first one, for example, is generated because there are some identical values in the weight vector, which is a bit problematic (but not critical) for the calculation of p-values.

Two-sample tests

Two sample tests work similarly to one-sample tests, but there are two samples instead of one and a preset hypothetical value. Let's create another sample vector, check the records for normality and compare them to the former weight vector:

```
weight2=c(72,74,76,78,79,80,81,81,82,83,87,89)
qqnorm(weight2)
qqline(weight2)
```



This QQ-plot is not as good as for the weight vector, so now it will make more sense to do the non-paramteric test as well but let's start it with the *two-sample t-test.*

```
> t.test(weight,weight2)

      Welch Two Sample t-test

data:  weight and weight2
t = -3.4875, df = 13.284, p-value = 0.003892
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -21.158063  -4.993453
sample estimates:
mean of x mean of y
 67.09091  80.16667
```

The difference between the sample means is nearly 13 kg, so we can expect a highly significant result, and we did receive it in the form of a large t-value (in absolute value) and a very small p-value, so the difference is real, the samples come from different populations.

The non-parametric version is called the *two-sample Wilcoxon test*, also known as the *Mann-Whitney test*:

```
> wilcox.test(weight,weight2)

      Wilcoxon rank sum test with continuity correction

data:  weight and weight2
W = 23, p-value = 0.008837
alternative hypothesis: true location shift is not equal to 0

Warning message:
In wilcox.test.default(weight, weight2) :
  cannot compute exact p-value with ties
```

Although the sample sizes are still small, the large difference in the sample means was enough to receive significant result with the non-parametric test as well. At this point we also mention that non-parametric tests also have some assumptions; e.g. Mann-Whitney requires that the two groups have identical distributions (the probability functions or density functions should look similar). This is not checked here, but there are methods to assess this feature, too (see. e.g. the Kolmogorov-Smirnov test).

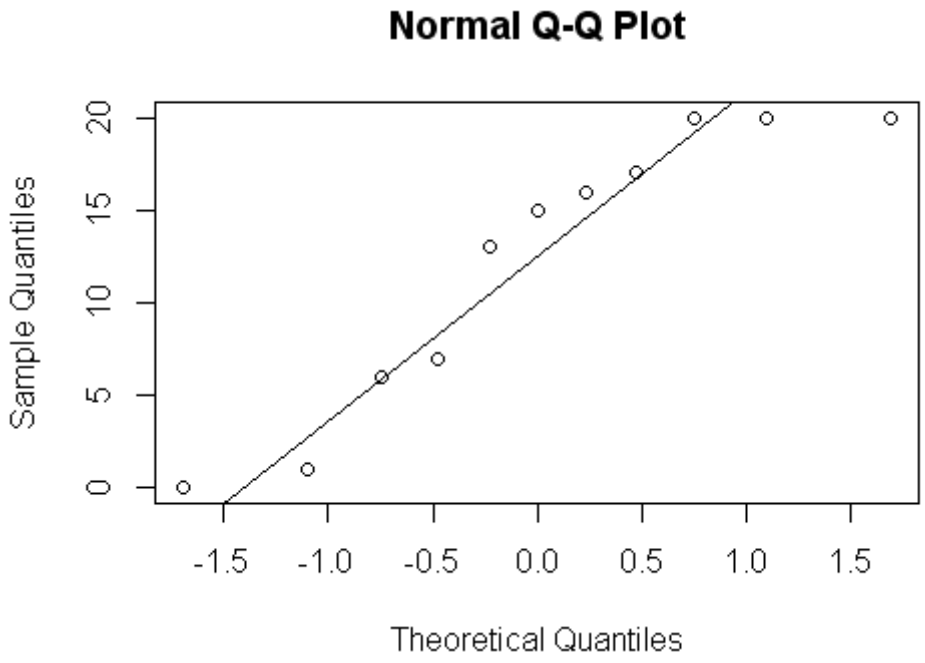
Paired tests

Sometimes records in two samples are not independent but each record in one sample has a pair in the other sample, like before treatment and after treatment values of the same set of subjects. Paired tests are developed exactly for these applications. In these tests, the difference has to follow normal distribution, not the individual samples. The samples, of course, need to be of the same length.

```
> length(weight)
[1] 11
> length(weight2)
[1] 12
> weight3=weight2[1:11]
```

The weight2 vector was longer than the weight vector, so we removed the last record. This procedure is only needed for getting data for illustration; no such manipulation of data is to be done in real-life situations.

```
> qqnorm(weight3-weight)
> qqline(weight3-weight)
```



Apart from the largest difference (the rightmost point), most of the points are close to the qq-line; so, we can proceed to the parametric test, the *paired t-test*.

```
> t.test(weight3,weight,paired=T)
```

```
Paired t-test

data:  weight3 and weight
t = 5.4, df = 10, p-value = 0.0003014
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  7.208775 17.336679
sample estimates:
mean of the differences
               12.27273
```

The paired=T argument of the t.test() function specifies the paired nature of the samples. We received a highly significant result, so the repeated measurement confirms some gain of weight in the patients, e.g. as a result of some a side-effect of a study medication. The t-value is positive, indicating the gain. Be careful to order the samples correctly in the script: the before weight has to be subtracted from the after weight.

If the differences do not follow normal distribution, proceed to the paired version of the two-sample Wilcoxon test:

```
> wilcox.test(weight3,weight,paired=T)

        Wilcoxon signed rank test with continuity correction

data:  weight3 and weight
V = 55, p-value = 0.005793
alternative hypothesis: true location shift is not equal to 0

Warning messages:
1: In wilcox.test.default(weight3, weight, paired = T) :
  cannot compute exact p-value with ties
2: In wilcox.test.default(weight3, weight, paired = T) :
  cannot compute exact p-value with zeroes
```

The test yielded significant result, so we can again state that the after weights are significantly higher than the before weights (V=55, p=0.006). Ignore the warning messages.

Test for variances

All tests we have discussed so far test middle values of the sample(s) and thus give information on middle values of the populations. However, other parameters of the population can also be of interest. Let's take the example of temperature stability in a cell incubator. The datasets below are measured internal temperatures of an incubator when set to 24°C and 3°C:

```
> temp24=c(25,26,23,23,25,24,24,25,24,25,23,24,25)
> temp3=c(3,1,2,6,3,5,1,0,6,7,3,2)
> var(temp24)
[1] 0.8974359
> var(temp3)
[1] 5.113636
```

The variances greatly differ but is this a real difference or just caused by random chance? This can be tested with an *F-test* using the var.test() function of R:

```
> var.test(temp24,temp3)

        F test to compare two variances

data:  temp24 and temp3
F = 0.1755, num df = 12, denom df = 11, p-value = 0.005604
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.05117154 0.58291524
sample estimates:
ratio of variances
         0.1754986
```

The test confirms that the variances are different, meaning that the internal temperature stability of the incubator is lower on 3°C then on 24°C. If you need to incubate at low temperature, avoid purchasing this instrument but it seems OK for ambient temperature incubation.

SUMMARY

- One sample vs. hypothetical value → one-sample t-test or Wilcoxon signed-rank test
- Two independent samples → two-sample t-test or Mann-Whitney test
- Paired samples → Paired t-test or paired Wilcoxon test
- Test variances: F-test

R functions of Chapter 4

t.test

wilcox.test

var.test

Chapter 5

Correlation

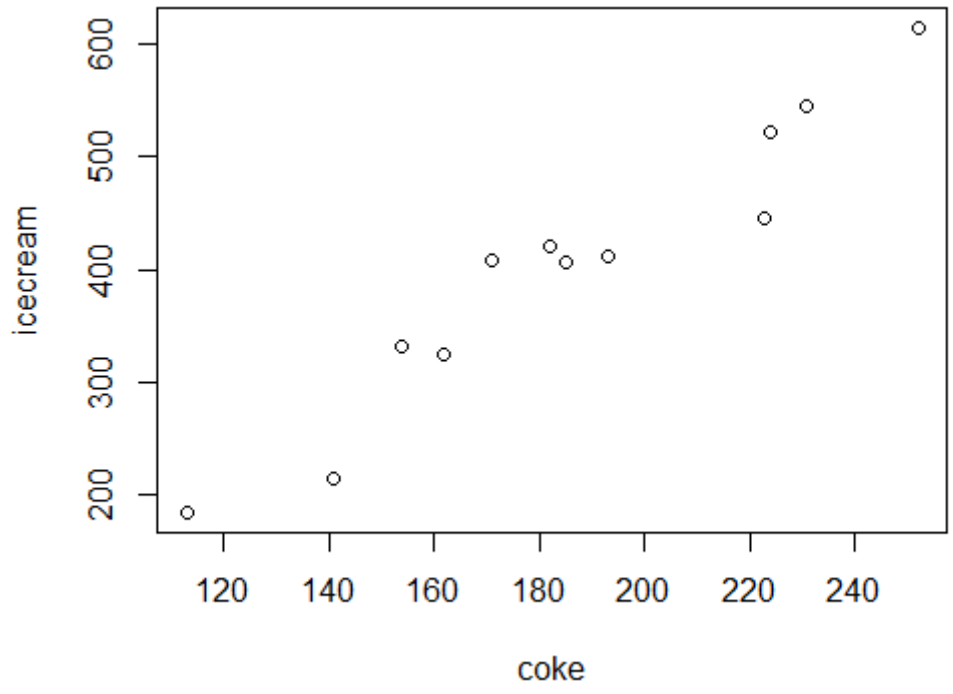
Correlation is a symmetric linear relationship between two or more variables. This means, that if there is correlation between variable A and B, variable A will increase/decrease gradually as variable B increases/decreases. It can be positive, if the sign of the changes is the same but when it is opposite, we call it negative correlation. Correlation is symmetrical because we do not assume that the values of one variable drive the values of the other; so, correlation is about changing together. Let's see an example, the daily sales of two goods in a store:

```
> coke=c(141,162,113,154,185,224,193,252,231,182,223,171)
> icecream=c(215,325,185,332,406,522,412,614,544,421,445,408)
```

Obviously, data are paired, meaning that the *i*th record in each sample belong to the same day. This also means that vectors to be correlated must be of the same length.

Plotting the values suggest that on the days when more coke is sold, customers tend to buy more ice-cream as well, and vice versa:

```
> plot(coke,icecream)
```



The exent of correlation, i.e. the measure of how closely the variables follow each other can be given with the *Pearson's correlation coefficient*, abbreviated as "r". It ranges between -1 and 1. r=-1 mean complete negative correlation, r=1 is complete positive correlation; while r=0 means that there is no linear relationship between the variables whatsoever. An r with a high absolute value is always good but be careful. If it is too high in a real biological system, it may mean that the variables not only correlate, but can be calculated from each other, such

as the height of a tree and the length of its shadow. Pearson's correlation coefficient can be calculated with the cor() function:

```
> cor(coke,icecream)
[1] 0.9649818
```

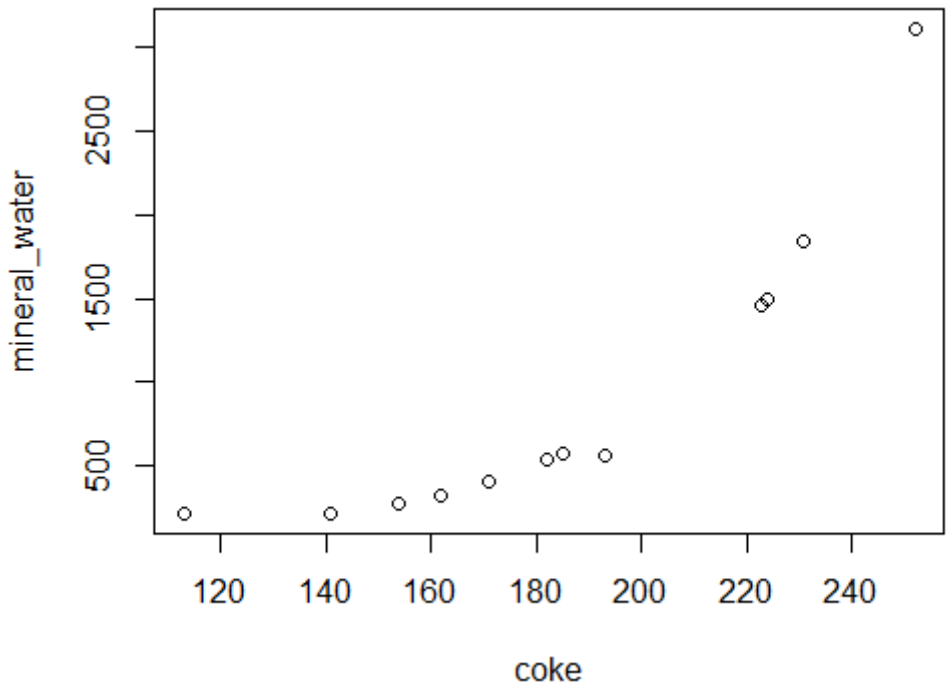
It seems there is a strong correlation between the sales of the two goods. However, we need to confirm that the difference from 0 is not caused by random chance. For this, a hypothesis test can be done, similar to the tests discussed in Chapter 4. The test is the *correlation test* and implemented with the cor.test() function:

```
> cor.test(coke,icecream)

        Pearson's product-moment correlation

data:  coke and icecream
t = 11.633, df = 10, p-value = 3.91e-07
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.8764776 0.9903970
sample estimates:
      cor
0.9649818
```

The test statistic of the correlation test is a t-value. The p-value is expressed in an exponential form because it is so low. The 3.91e-07 expression means 3.91×10^{-7}. In a report or publication you need to phrase this result as "We found significant positive correlation between coke and ice-cream sales ($r=0.96$, $t=11.63$, $p<0.001$)".

There are, however, lots of relationships between data that are not linear, but follow exponential, logarithmic, logistic, power, etc. functions. In these cases linear correlation may not yield appropriate results, but you may need to transform your data to linear scale, if possible. Exponential and power functions can be transformed using logarithmic transformation. Let's see an example with the sales of a third item:

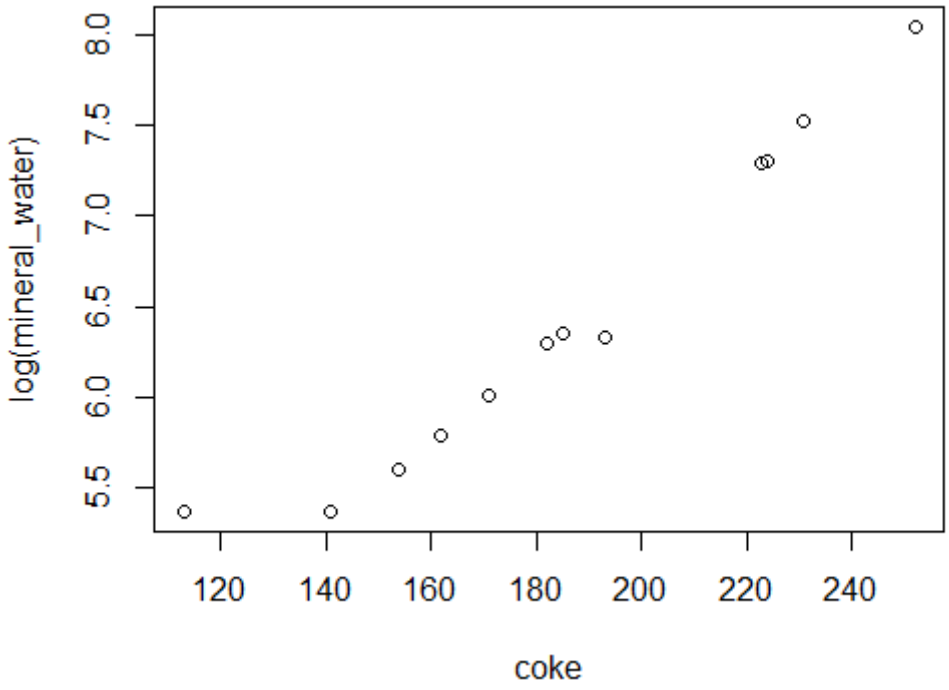
```
> mineral_water=c(215,325,215,272,576,1490,562,3114,1844,541,1460,408)
> plot(coke,mineral_water)
```




```
> cor(coke,mineral_water)
[1] 0.8817931
> cor.test(coke,mineral_water)

        Pearson's product-moment correlation

data:  coke and mineral_water
t = 5.9123, df = 10, p-value = 0.0001486
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.6233409 0.9665566
sample estimates:
      cor
0.8817931
```

The r-value is still rather high and it is significant as well, but this relationship does not seem linear but looks exponential. So, it is reasonable to transform it to linear by taking the logarithm of mineral water and perform the analysis again:

```
> plot(coke,log(mineral_water))
```




```
> cor(coke,log(mineral_water))
[1] 0.9748759
> cor.test(coke,log(mineral_water))

        Pearson's product-moment correlation

data:  coke and log(mineral_water)
t = 13.84, df = 10, p-value = 7.558e-08
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9102355 0.9931353
sample estimates:
      cor
0.9748759
```

The log() function calculates the logarithm of the parenthetic part. The default base is *e*, so it calculates natural logarithm, but using the base argument, this can be changed (for details see the Help menu).

After logarithmic transformation, the r value became exceptionally good and the p-value also improved. The plot also indicates that the relationship is linearized.

As all tests, the correlations test also has its assumptions. One was linearity, which can be improved with transformations. If not, Pearson's correlation is not a good choice. Furthermore, normality of at least one of the variables is needed. This can again be checked with QQ-plot. Another assumption is *homoscedasticity*, which means that the variation of the data should not depend on the value of the data. As a graphical representation, this means that the data in the scatter plot should be arranged in a tube and not a cone. This can, however, be appropriately assessed if having a very high number of data. There are other measures of homoscedasticity but we will discuss these in Chapter 6. If data are not homoscedastic, they are *heteroscedastic*.

If any of the assumptions is severely violated or the scale of the variables is not numeric but ordinal, or you cannot/do not want to transform your data, you still have alternatives: You can calculate *rank-based correlations*, either *Spearman's rho* or *Kendall's tau* instead of Pearson's r. According to contemporary literature, Kendall is preferred but, basically both are good for the purpose. Since these are rank-based correlations, most transformations do not affect their results, which we can easily confirm with the available data:

```
> cor.test(coke,mineral_water,method="kendall")

        Kendall's rank correlation tau

data:  coke and mineral_water
z = 4.3303, p-value = 1.489e-05
alternative hypothesis: true tau is not equal to 0
sample estimates:
      tau
0.9618601

Warning message:
In cor.test.default(coke, mineral_water, method = "kendall") :
  Cannot compute exact p-value with ties

> cor.test(coke,log(mineral_water),method="kendall")

        Kendall's rank correlation tau

data:  coke and log(mineral_water)
z = 4.3303, p-value = 1.489e-05
alternative hypothesis: true tau is not equal to 0
sample estimates:
      tau
0.9618601

Warning message:
In cor.test.default(coke, log(mineral_water), method = "kendall") :
  Cannot compute exact p-value with ties
```

Note that the untransformed and transformed data indeed lead to the same results. The function used for non-parametric (rank-based) correlation is the same as for the parametric one (Pearson's). This can be possible because the default method of the cor.test() function is the Pearson's; therefore we did not have to specify it. If you prefer Spearman's rho, the method argument has to be set to "spearman":

```
> cor.test(coke,log(mineral_water),method="spearman")

        Spearman's rank correlation rho

data:  coke and log(mineral_water)
S = 2.5039, p-value = 3.992e-10
alternative hypothesis: true rho is not equal to 0
sample estimates:
     rho
0.991245

Warning message:
In cor.test.default(coke, log(mineral_water), method = "spearman") :
  Cannot compute exact p-value with ties
```

SUMMARY

Symmetric relationship between variables → Correlation

Pearson's correlation

If assumptions not met → Spearman's rho or Kendall's tau

R functions of Chapter 5

cor
cor.test

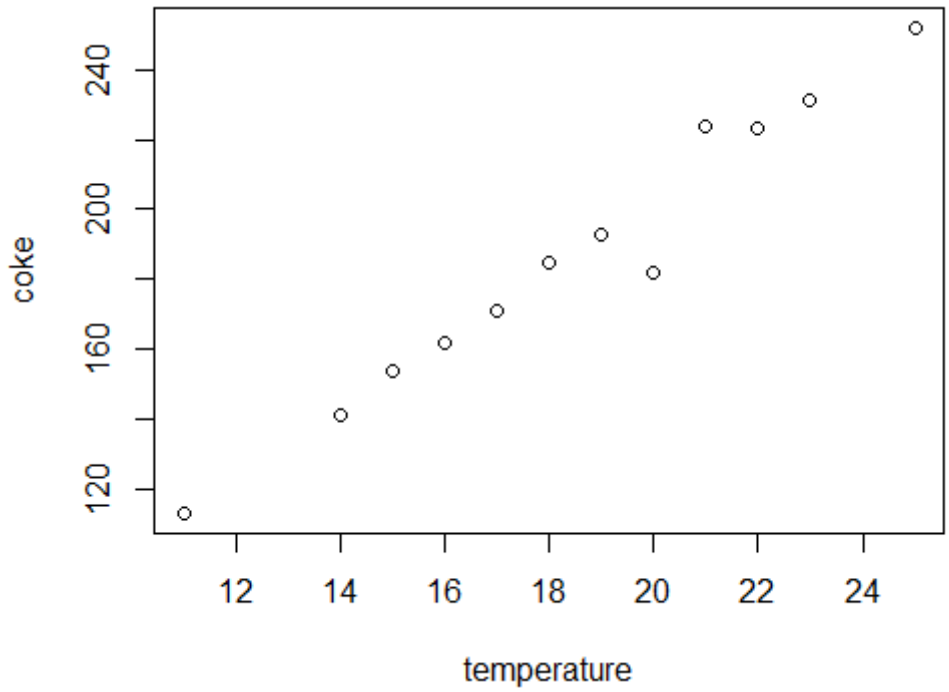
Chapter 6

Linear regression

Simple linear regression

Unlike correlation, *regression* implies cause–effect relationship between the variables, meaning that one of the variables drives the other. The former is called a *predictor*, an *explanatory variable* or an *independent variable* (these all mean the same thing), while the latter is called a *response variable* or a *dependent variable*. So, an increase in the predictor can CAUSE the response variable to increase or decrease. The sales of the mentioned goods are good examples of response variables, whereas a potential predictor can be air temperature. If it goes up, people buy more ice-cream and drink more coke and water. So, for the difference between a correlation and a regression is only conceptual but regression is a modelling approach, so it also gives a model for the relationship, which in turn can be used for a variety of purposes, such as to estimate unknown sales for new temperature values.

```
> temperature=c(14,16,11,15,18,21,19,25,23,20,22,17)
> plot(coke~temperature)
```



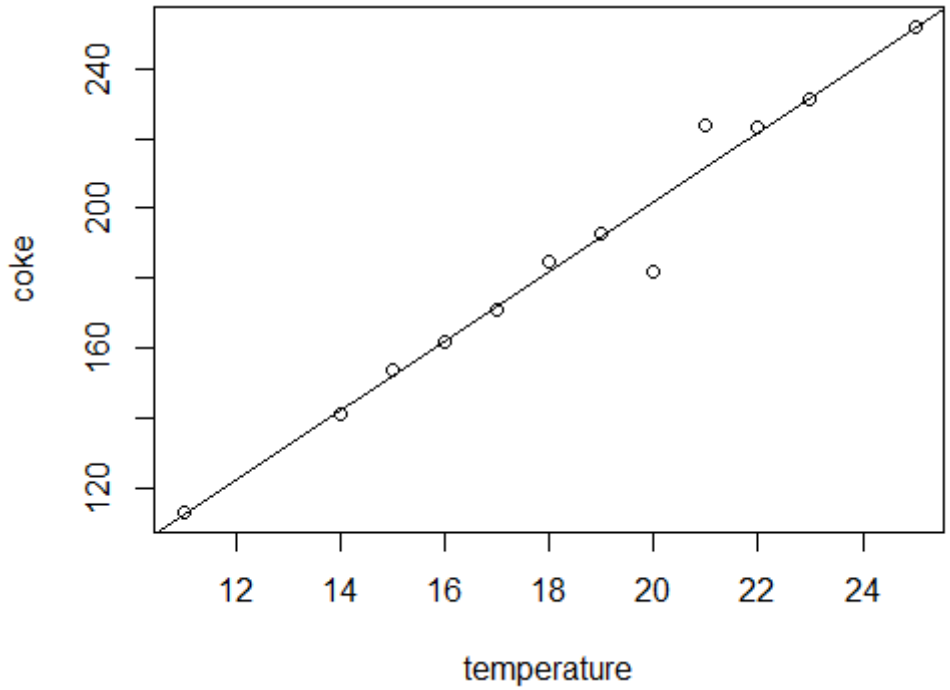
Since we assume cause-effect relationship, we used a formula to express it when plotting. Be careful to arrange the variables in the correct order; the parenthetic part means "coke as a function of temperature". This is a linear relationship, so we can express it as a linear function in the following form: *response variable*= $beta_0$ + $beta_1$ × *predictor* + *error*,

where $beta_0$ is the value where the model line intercepts the y-axis, $beta_1$ is the steepness of the model line and the model values of the response variable deviate from the expected value

due to some measurement or other types of error. Such a model line can be calculated for the coke~temperature relationship with the lm() function (linear model):

```
> mod1=lm(coke~temperature)
> mod1

Call:
lm(formula = coke ~ temperature)

Coefficients:
(Intercept)  temperature
      2.844        9.941
```

The lm() function creates a model type of object in the environment. The function call gave estimates for the two coefficients ($beta_0$ and $beta_1$). So, we can see that at 0°C, people by only 2.8 cokes a day on average and every degree of Celsius increase in temperature entails an increase of 9.9 in the daily coke sales. The model line defined with these parameters can be added to the scatterplot with the `abline(mod1)` script:



The model line is generated by R by minimizing the sum of squared distances between modelled and empirical response variable values for each temperature value. This difference appears in the general equation as the "error" and once having the model, we call these *residuals*.

The model can be tested for significance with the summary() function:

```
> summary(mod1)

Call:
lm(formula = coke ~ temperature)
```

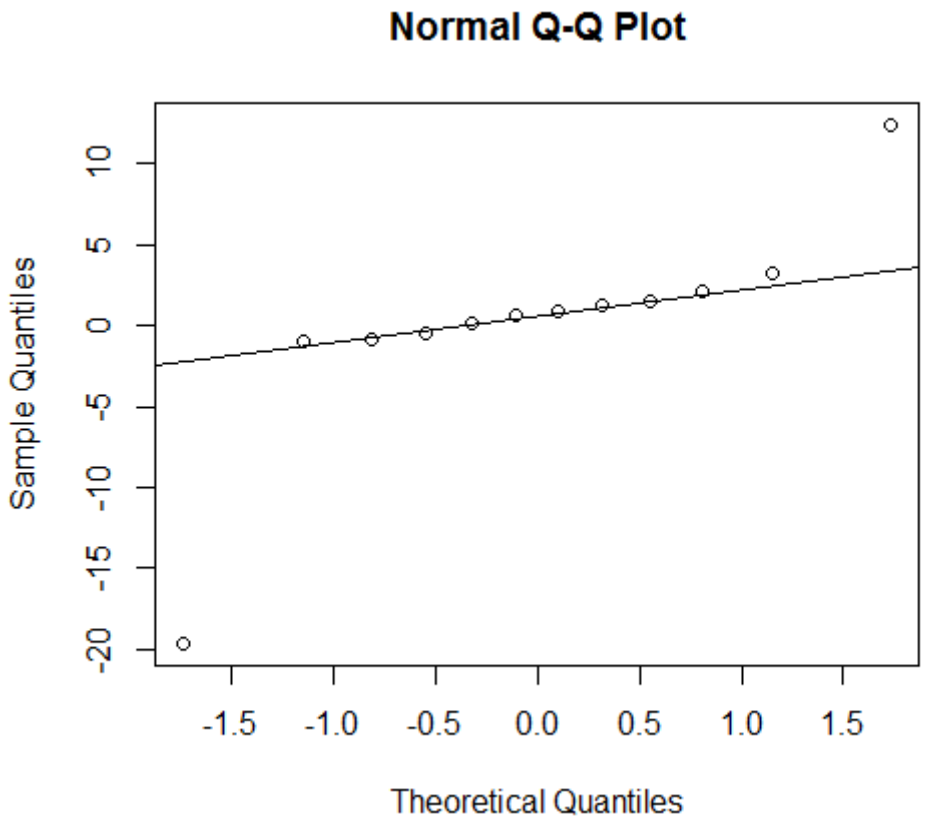
```
Residuals:
     Min       1Q   Median       3Q      Max
-19.6559  -0.5668   0.7252   1.6089  12.4035

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   2.8443    10.4866   0.271    0.792
temperature   9.9406     0.5572  17.842 6.53e-09 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 7.494 on 10 degrees of freedom
Multiple R-squared:  0.9695,   Adjusted R-squared:  0.9665
F-statistic: 318.3 on 1 and 10 DF,  p-value: 6.534e-09
```

This output contains a summary of the residuals (like the summary of a vector) and some test results. Both parameters are tested for significance: the test of the intercept is not interesting now but that of the temperature is. It has a high t-value and a low p-value, indicating that the slope of the model line differs from zero (i.e. from a horizontal line with no relationship), meaning that temperature has an effect on coke sales. An adjusted R^2*-value* (*coefficient of determination*) is also calculated, which equals the proportion of the variation of the coke data explained by the model. This is 0.97, which now means that 97% of the variation is explained by temperature. This proportion of the variation is also tested for significance using an F-test. Since temperature is the only predictor in the model, the corresponding p-value equals the p-value received for the slope. However, when there are more predictors in the model, these p-values differ (see later).

When having a linear model and tested it for significance, you can report it by writing “we showed that temperature has a significant positive linear effect on coke sales (t=17.84, $p<0.001$, $R^2=0.967$)”.

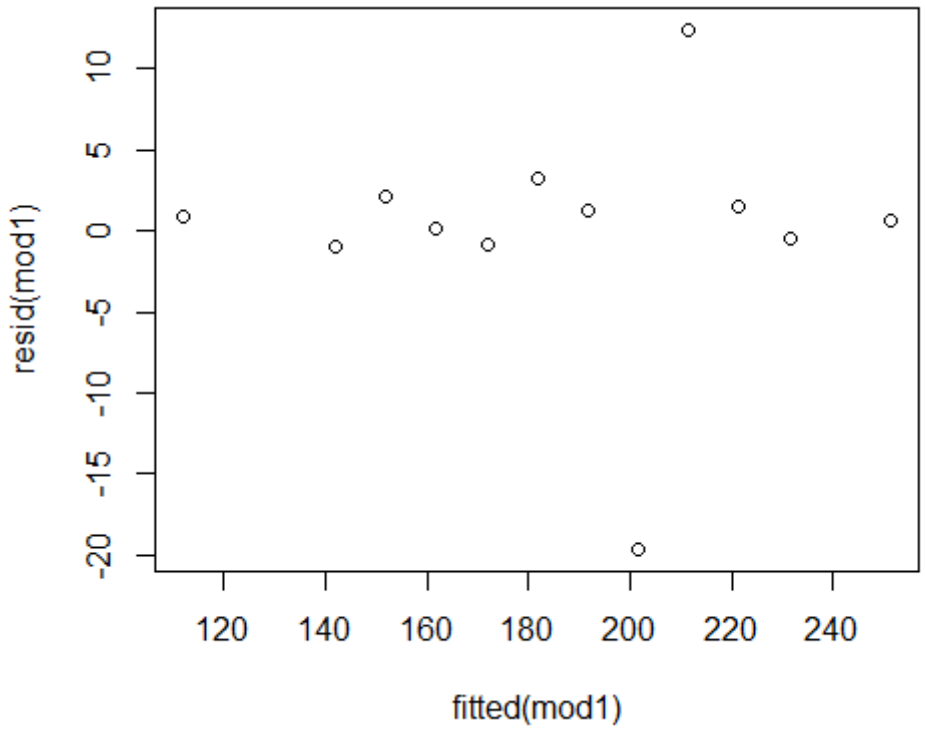
Linear modelling also has some assumptions on the data. The residuals need to follow normal distribution and they must be independent from the predictor. These can be checked visually. The first with QQ-plot, the other by a simple scatterplot. Residuals can be extracted from a model object by the resid() function.

```
> qqnorm(resid(mod1))
> qqline(resid(mod1))
```



Residuals, apart from the smallest and largest ones, pretty much follow normal distribution. You may consider inspecting and removing the corresponding coke sales...

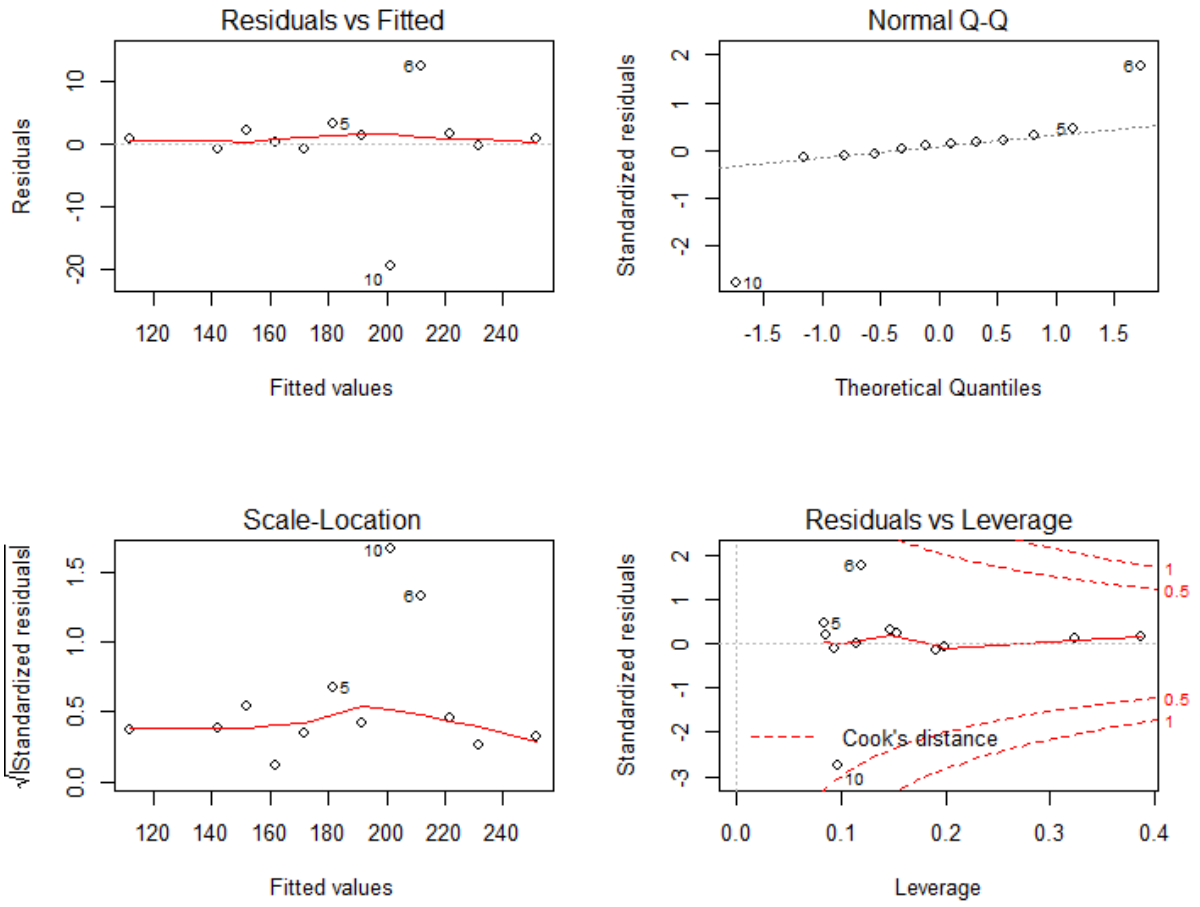
```
> plot(resid(mod1)~fitted(mod1))
```



This scatterplot shows whether the residuals depend on the value of coke sales. Since points are aligned horizontally, there is no need to worry; residuals are independent from the fitted values. There are two points that scatter farther away but lend no alarming structure to the scatterplot.

These diagnostic plots can also be inquired simply by the `plot(mod1)` script. There are four plots to show, the first two being the essential ones we prepared manually above, the rest are not so important now. After running the script, R will prompt you to press Enter in the console, so move the cursor to the console and press enter to display the plots. In this case it can be useful to temporarily split the plot window into a 2 by 2 grid to display all figures at once but remember to set it back to the original 1 by 1 layout:

```
> par(mfrow=c(2,2))
> plot(mod1)
> par(mfrow=c(1,1))
```

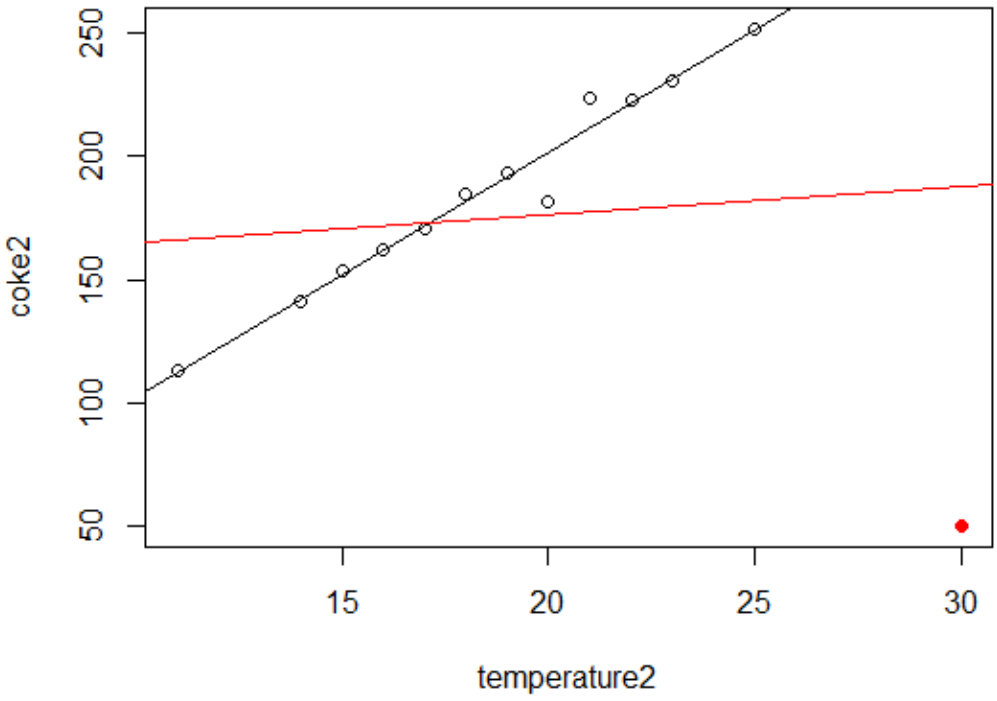


The dashed line in the first plot indicates complete independence between residuals and fitted values. The model residuals fit this line rather well, as suspected from the manually prepared plot. The second plot is identical to the QQ-plot we prepared.

Not related to assumptions, but there is one more thing to consider regarding the reliability of regression models. This is the *leverage*, i.e. the effect of individual points on the regression outcome. If a point has a high leverage, it is considered an influential point. These should be

inspected because they can distort the results. The most common problem is to have an influential point at the lower or upper extreme of the predictor's scale. These, like levers, can attract the model line, leading to false results. The farther these points are from the gravity center of the points, the higher their effect. Including such an influential point in the original temperature and coke vectors modify the model considerable:

```
> plot(coke~temperature)
> abline(mod1)
> points(30,50,col="red",pch=16)
> coke2=c(coke,50)
> temperature2=c(temperature,30)
> mod2=lm(coke2~temperature2)
> abline(mod2)
> summary(mod2)

Call:
lm(formula = coke2 ~ temperature2)

Residuals:
     Min       1Q   Median       3Q      Max
-137.571  -16.583    5.754   44.489   70.092

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   153.595     64.409   2.385   0.0362 *
temperature2    1.133      3.236   0.350   0.7329
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 56.49 on 11 degrees of freedom
Multiple R-squared:  0.01101,  Adjusted R-squared:  -0.07889
F-statistic: 0.1225 on 1 and 11 DF,  p-value: 0.7329
```



As you can see, adding a coke sale of 50 to temperature 30°C tilts the model line considerably and renders the model non-significant (t=0.35, p=0.733, R^2=-0.79). Obviously, this outlier

should not be considered for modelling; it may by caused by the fact that when it gets really hot (30°C), people simply go down to the beach and buy their coke there instead of the store whose sales we analyze. It is the responsibility of the researcher and peers during the peer review process to identify such sources of misinterpretation.

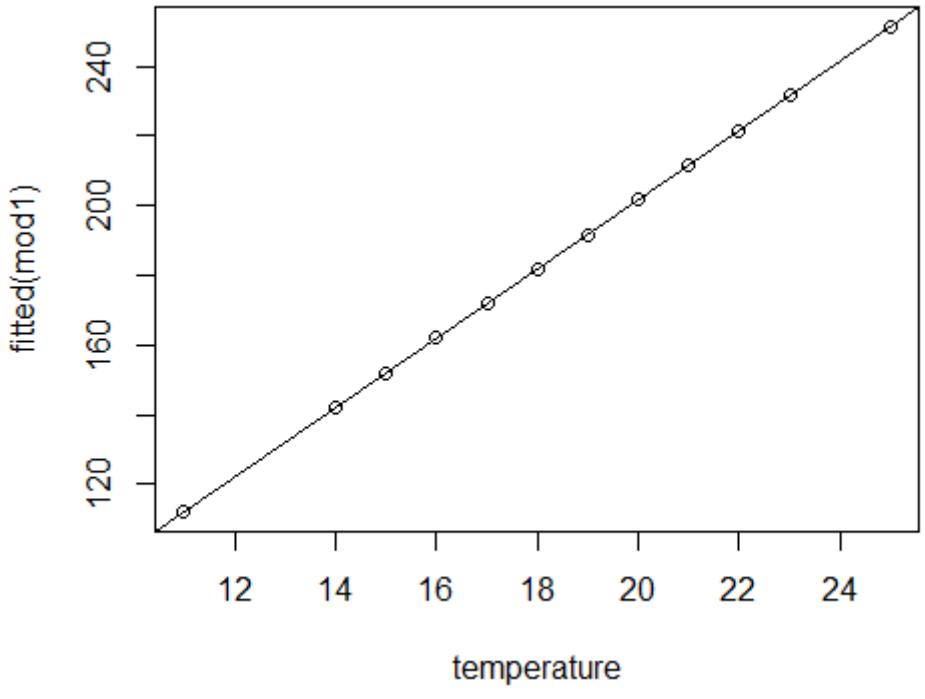
Predictions with linear models

Using a linear model, several further details can be extracted. The fitted coke sales, i.e. the values that we would have had without random/unknown effects on sales (equivalent to the error or the residuals), can be calculated with the fitted() function:

```
> fitted(mod1)
       1        2        3        4        5        6        7        8
142.0124 161.8936 112.1907 151.9530 181.7748 211.5965 191.7153 251.3588
       9       10       11       12
231.4777 201.6559 221.5371 171.8342
```

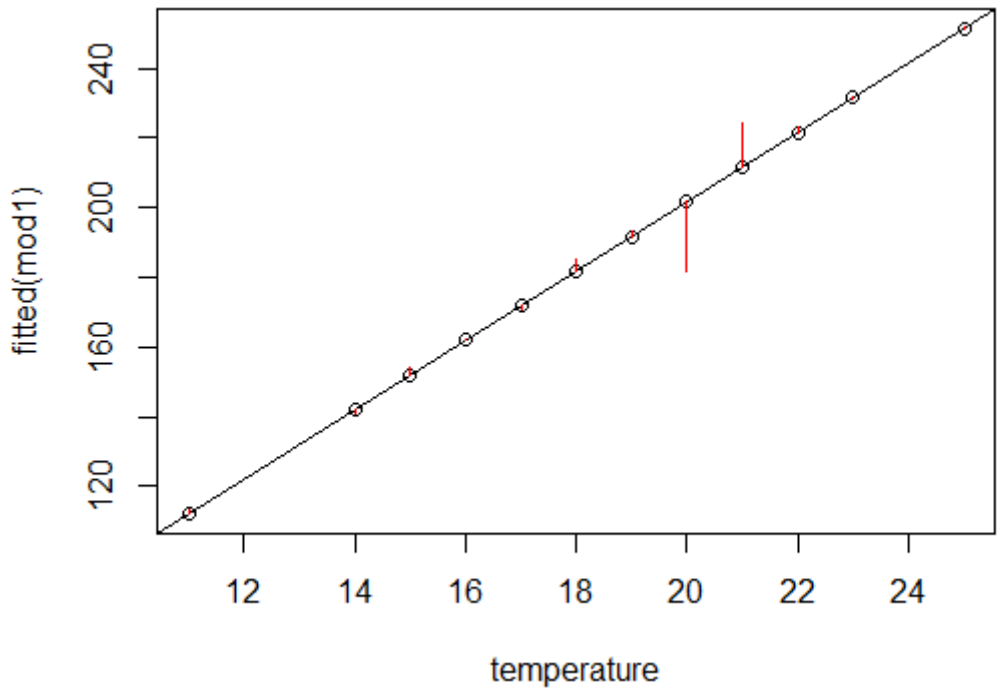
The order corresponds to the order of records in the coke vector. By definition, these values fit the model line perfectly:

```
> plot(fitted(mod1)~temperature)
> abline(mod1)
```



The residuals can be easily added to the graph using the segments() function. For segments, you need to provide the x and y coordinates of the starting and ending points of the segments, that is, you need to provide four vectors with identical lengths:

```
> segments(temperature,coke,temperature,fitted(mod1),col="red")
```



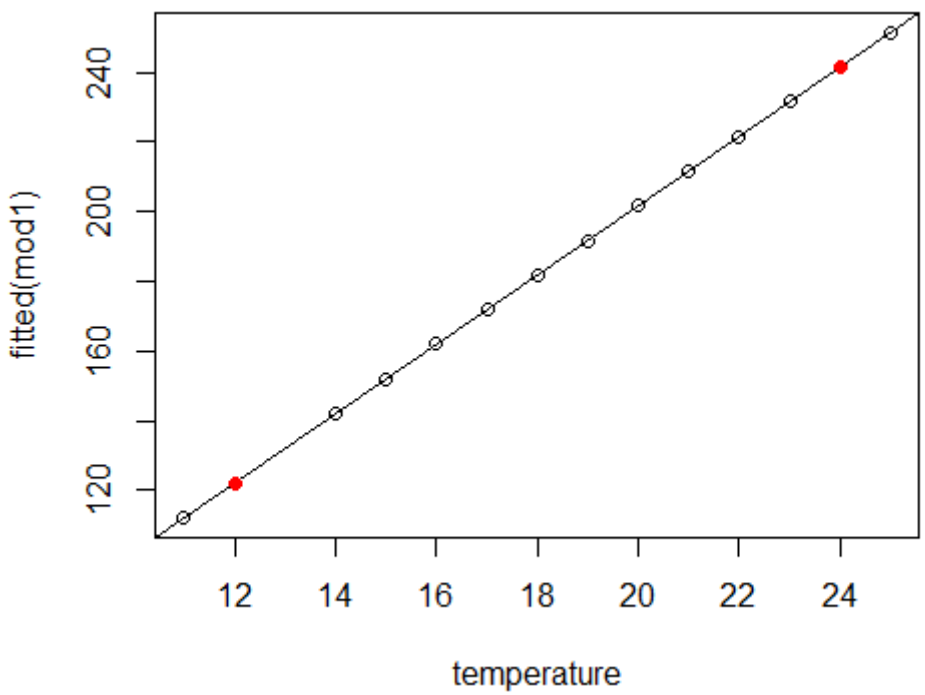
In all but two cases the errors were negligible. For better visibility, the segments are in red.

Predicting coke sales for new temperature values can easily be done using the model. First, you need to create a data frame, containing the new temperature values you would like to calculate coke sales for. Let these temperatures be 12 and 14°C:

```
> new_temperature=data.frame(temperature=c(12,24))
> new_temperature
  temperature
1          12
2          24

> predict(mod1,newdata=new_temperature)
       1        2
122.1313 241.4182
```

So, the predicted coke sales for 12 and 24°C are approx. 122 and 241, respectively. These can be placed on the original line to check whether they are correctly calculated. The predicted values need to be stored in a vector and the plot has to be redrawn:

```
> new_coke=predict(mod1,newdata=new_temperature)
> plot(fitted(mod1)~temperature)
> abline(mod1)
> points(c(12,24),new_coke,col="red",pch=16)
```



The new coke sales appear as full circles in red.

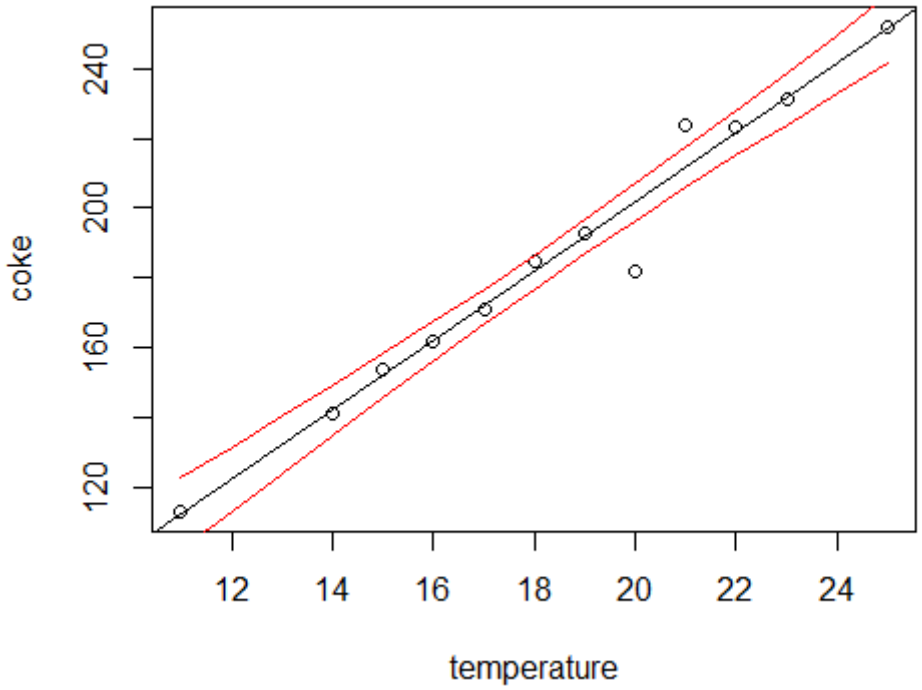
Using the model, you can predict some further very useful things, like *confidence bands* and *prediction bands*. A confidence band will include the real linear relationship line of the total population with a preset confidence (95% as default), while the prediction band will include all future records with a preset probability (again, 95% as default).

First, you need start a new plot, define the temperature interval you would like to calculate the bands (this should not extend over the smallest and largest predictor values you have response values for!). Coordinates for bands can be predicted as for new temperature values, the only difference being that you need to specify you are interested in the confidence or prediction bands using the interval argument. The output of this prediction will include the lower and upper band limits and the fitted values. You can to add the band limits to the plot using the lines() function:

```
> plot(coke~temperature)
> abline(mod1)
> predictframe2=data.frame(temperature=11:25)
> fitted2=predict(mod1,newdata=predictframe2,interval="confidence")
> fitted2
        fit      lwr      upr
1  112.1907 101.7980 122.5834
2  122.1313 112.8206 131.4420
3  132.0719 123.7983 140.3454
4  142.0124 134.7119 149.3130
5  151.9530 145.5323 158.3737
6  161.8936 156.2160 167.5712
7  171.8342 166.7031 176.9652
8  181.7748 176.9268 186.6227
9  191.7153 186.8410 196.5897
10 201.6559 196.4503 206.8615
11 211.5965 205.8069 217.3861
12 221.5371 214.9779 228.0963
13 231.4777 224.0205 238.9348
```

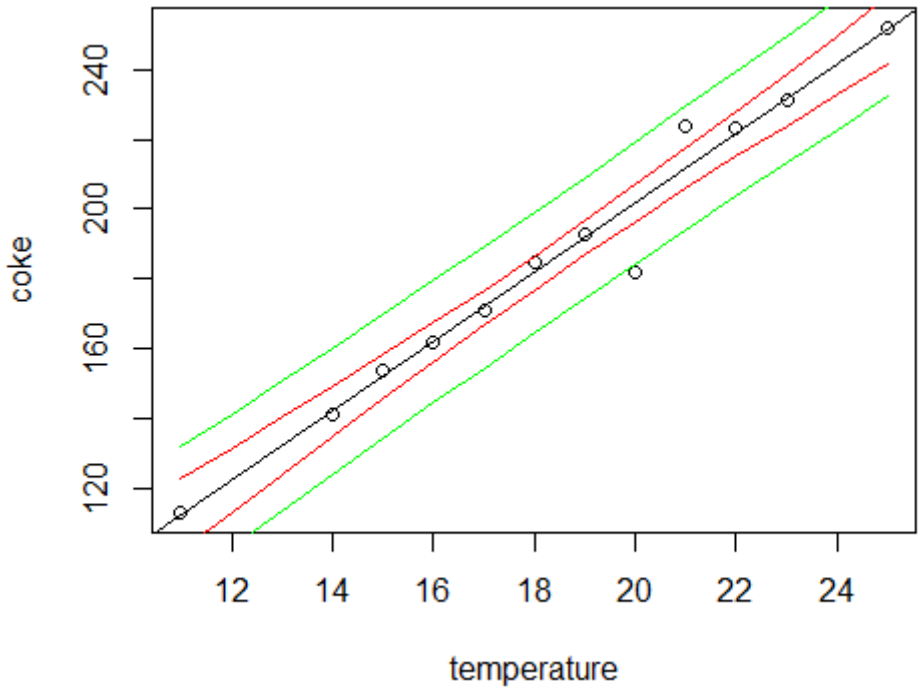
```
14 241.4182 232.9756 249.8609
15 251.3588 241.8705 260.8471
```

```
> lines(11:25,fitted2[,"lwr"],col="red")
> lines(11:25,fitted2[,"upr"],col="red")
```



The band is rather narrow, so the real relationship is close to the model line. The prediction band can be added similarly. It is always a wider band.

```
> fitted3=predict(mod1,newdata=predictframe2,interval="prediction")
> lines(11:25,fitted3[,"lwr"],col="green")
> lines(11:25,fitted3[,"upr"],col="green")
```



Multiple linear regression

The measurable properties of biological objects can be determined by several different factors, thus it may be necessary to include more than one of predictors into linear regression models. Body weight, e.g. is determined by the diet, age, genetic background, etc. of individuals. In experimental designs, control groups and treatment groups are frequently selected so that they differ only in the studied variable, but sometimes this is not possible or, in other cases, we may be interested in the complex effect of multiple drivers at the same time. In such cases, *multiple linear regressions* are done, provided all variables are numeric and are defined on interval or absolute scales.

The following data are based on an antelope reproduction study. The response variable is the annual number of newborn animals, and it is modeled using the total antelope population each year, the annual precipitation and the annual mean temperature. In real-life situations datasets are imported from external files but now we have to type the data in:

```
> fawn=c(290,240,180,255,300,160,340,210)
> adult=c(920,870,720,850,960,680,970,790)
> prec=c(335,292,274,312,320,269,358,284)
> temp=c(23.1,25.2,23.7,21.9,23.5,22.3,26.4,25.3)
```

Predictors are separated from each other in the formula with the + sign:

```
> multimod=lm(fawn~adult+prec+temp)
> summary(multimod)

Call:
lm(formula = fawn ~ adult + prec + temp)

Residuals:
      1       2       3       4       5       6       7       8
-7.1462 -3.1135  1.7719  2.3409  4.1871  0.1939  2.6177 -0.8518

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -325.15584   31.29663 -10.389 0.000485 ***
adult          0.31031    0.04098   7.572 0.001630 **
prec           0.94122    0.13746   6.847 0.002381 **
temp           0.93104    1.26299   0.737 0.501922
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 4.861 on 4 degrees of freedom
Multiple R-squared:  0.9965, Adjusted R-squared:  0.9938
F-statistic: 377.3 on 3 and 4 DF,  p-value: 2.322e-05
```

According to the output, the total number of adult antelopes is an important determinant of the number of fawns but precipitation is also a significant predictor; probably because if there is more precipitation, more food is available for the animals. Temperature, however, does not affect the size of the new generation according to these data. We can also notice that the F-test of the full model has a much lower p-value than any of the individual predictors. The coefficient of determination is very high, so our model explains a very high proportion of the variation of the fawn data.

Predictors that turn out to have no effect on the response variable are unnecessary to include in the model, and they can even decrease the total explanatory power of the model. So, it is

advisable to sort all predictors out that do not increase the explained variance of the data. This can easily be done with the step() function, which removes superfluous variables one by one:

```
> multimod2=step(lm(fawn~adult+prec+temp))

Start:  AIC=27.76
fawn ~ adult + prec + temp

        Df Sum of Sq     RSS    AIC
- temp   1     12.84  107.37 26.775
<none>                 94.53 27.756
- prec   1   1108.02 1202.55 46.102
- adult  1   1355.14 1449.67 47.597

Step:  AIC=26.77
fawn ~ adult + prec

        Df Sum of Sq     RSS    AIC
<none>                107.37 26.775
- prec   1    1096.5 1203.90 44.111
- adult  1    1497.6 1604.94 46.411
```

The output shares some information of the stepwise process. The removal of unnecessary variables is based on some information criteria, but we do not discuss this in detail here. It is, however, clear that R started with the full model with all three predictors but then dropped temp and terminated the selection with two remaining predictors: the number of adults and the annual precipitation.

```
> summary(multimod2)

Call:
lm(formula = fawn ~ adult + prec)

Residuals:
      1       2       3       4       5       6       7       8
-8.2223 -2.2011  2.2059  0.4684  3.0926 -0.4321  4.4334  0.6552

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -306.5484    17.6372 -17.381 1.16e-05 ***
adult          0.3173     0.0380   8.351 0.000403 ***
prec           0.9338     0.1307   7.146 0.000834 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 4.634 on 5 degrees of freedom
Multiple R-squared:  0.996,   Adjusted R-squared:  0.9944
F-statistic: 622.6 on 2 and 5 DF,  p-value: 1.012e-06
```

The predictors that R retained still have significant effects but their p-values improved a bit and the p-value of the full model also improved. It should be noted that the decision of retaining or dropping a variable is not based on its significant effect but on whether it improves the explanatory power of the full model, so sometimes non-significant predictors are also retained. This is normal; do not be surprised to see this in a future analysis of yours.

Multiple linear regression has the same assumptions as simple linear regressions but besides these, the predictors must also be independent of each other, meaning that they should not correlate with each other. This is often phrased as “there should be no *multicollinearity* between predictors. Correlating predictors carry the same information, so including both means the inclusion of the same information twice. Therefore, if predictors correlate, only one of them can be used for modeling. The correlation of the predictors can be checked by calculating and testing correlation between each pair but this can also be done quickly with

the cor.table() function of the *picante* package. The package has to be installed and then loaded in the R workspace. The function can work with data frames, so we need to make a data frame from the variables. It is enough to include only the predictors in the data frame:

```
> multi_data=data.frame(adult,prec,temp)
> head(multi_data)
  adult prec temp
1   920  335 23.1
2   870  292 25.2
3   720  274 23.7
4   850  312 21.9
5   960  320 23.5
6   680  269 22.3

> cor.table(multi_data)

$r
          adult      prec      temp
adult 1.0000000 0.9035397 0.3725939
prec  0.9035397 1.0000000 0.3077186
temp  0.3725939 0.3077186 1.0000000

$df
[1] 6

$P
             adult         prec      temp
adult 0.000000000 0.002084612 0.3633508
prec  0.002084612 0.000000000 0.4584155
temp  0.363350785 0.458415528 0.0000000
```

The output is a list of three items; individual items are marked with the $ sign. The $ sign can be used to extract only certain parts of such outputs in the same way as a variable is specified in a data frame.

It seems that the number of adults and precipitation correlates; if there is more precipitation, there are more adults. So, the ultimate driver of fawn number is probably precipitation and our multimod model is inflated with the same information from precipitation and adult number. The most correct model, thus, includes only precipitation as the predictor.

Another method to identify multicollinearity is calculating the VIF (variance inflation factor) using the vif() function of the faraway package. The function allocates a VIF value to every predictor. If there is at least one predictor with a VIF above 4, there is considerable multicollinearity. If you have such a variable, remove it and prepare the model again and check again for remaining multicollinearity. So, remove multicollinearity step by step by removing one predictor at a time. Variance inflation factor also considers multiple correlations, so it is more powerful than simply checking pair-wise correlations. It is advisable to check for VIF before variable selection:

```
> vif(multimod)
   adult     prec     temp
5.755740 5.475137 1.167387
```

Note also that the function makes the calculation using the model and not the raw variable vectors. The number of adults has the highest VIF, so it has to be removed and then multicollinearity has to be checked again:

```
> multimod3=lm(fawn~prec+temp)
```

```
> vif(multimod3)
    prec     temp
1.104595 1.104595
```

There is no more multicollinearity between the remaining variables. Now we can do the model selection again:

```
> multimod4=step(multimod3)
Start:  AIC=47.6
fawn ~ prec + temp

       Df Sum of Sq     RSS    AIC
- temp  1     155.3  1604.9 46.411
<none>               1449.7 47.597
- prec  1   21707.2 23156.8 67.765

Step:  AIC=46.41
fawn ~ prec

       Df Sum of Sq     RSS    AIC
<none>               1604.9 46.411
- prec  1     25242 26846.9 66.948

> summary(multimod4)

Call:
lm(formula = fawn ~ prec)

Residuals:
    Min      1Q  Median      3Q     Max
-16.799  -9.130  -5.376   8.063  25.286

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -339.6556    60.6549  -5.600  0.00138 **
prec           1.9199     0.1976   9.714 6.83e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 16.36 on 6 degrees of freedom
Multiple R-squared:  0.9402, Adjusted R-squared:  0.9303
F-statistic: 94.37 on 1 and 6 DF,  p-value: 6.832e-05
```

The selection removed temperature from the model, and the final model contained only precipitation as the sole predictor. The corresponding p-value is the best so far, so we can be satisfied with the result.

SUMMARY

Linear cause–effect relationship → Linear regression

Usage:

- Testing the effect of a predictor (beware of influential points with high leverage)
- Predictions
    - New data
    - Confidence band
    - Prediction band

More than one potential predictor → Multiple linear regression

Improving the model:

- Ruling out multicollinearity (pairwise correlations or VIF)
- Variable selection

R functions of Chapter 6

lm

abline

summary

resid

fitted

segments

predict

points

lines

step

cor.table

vif

Chapter 7

Analysis of variance

T-tests and their non-parametric alternatives are used to compare a maximum of two samples. However, there may be need to compare three or more groups, like several different treatment groups and control groups. Theoretically, it would be possible to do all pairwise comparisons, but the chance for reaching inappropriate conclusion increases with the number of comparisons. In a simple pairwise comparison, we choose the H1, if the chance for it is at least 95%, which means that in 5% of the cases, we are wrong. If having e.g. five groups, we have to do 10 pairwise comparisons, leading to the accumulation of the error, as the chance for all results being correct as suggested by the p-values would be only 0.95^{10}=0.599. As a result, we approach multiple comparisons in a slightly different way, using a method called *ANOVA* (analysis of variance).

ANOVA is used to compare the means of three or more groups, or, with other words, it is used to test the effect of a nominal variable (whose levels are the groups) on continuous response variables. Basically, ANOVA compares the total variance of the data to the group variances. If these do not differ much, it is not possible that any of the groups have a different mean.

The data8.1.txt file (accessed through Coospace) contains fluorescence values of cells and their genotypes and age groups. Genotype is a nominal predictor of fluorescence; it has three levels: wild type, transgenic 1 and transgenic 2. Let's test its effect on fluorescence with ANOVA using the aov() function. The formulation of the test is very similar to that of linear models.

```
> cells=read.table("data8.1.txt",header=T)
> cells
   Fluorescence Genotype Age_group
1            21       WT         1
2            31       WT         1
3            33       WT         1
4            40       WT         2
5            41       WT         2
6            43       WT         2
7            52       WT         3
8            53       WT         3
9            60       WT         3
10           30   Trans1         1
11           42   Trans1         1
12           43   Trans1         1
13           49   Trans1         2
14           53   Trans1         2
15           53   Trans1         2
16           61   Trans1         3
17           64   Trans1         3
18           71   Trans1         3
19           52   Trans2         1
20           59   Trans2         1
21           64   Trans2         1
22           69   Trans2         2
```

```
23           71   Trans2         2
24           73   Trans2         2
25           81   Trans2         3
26           84   Trans2         3
27           89   Trans2         3
```

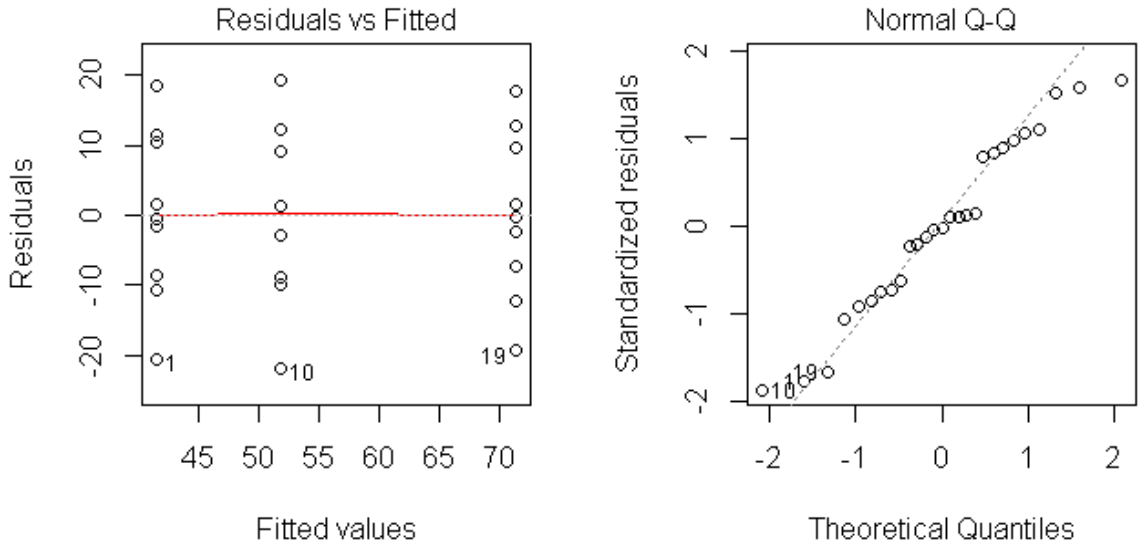
```
> anova1=aov(Fluorescence~Genotype,data=cells)
> summary(anova1)
            Df Sum Sq Mean Sq F value   Pr(>F)
Genotype     2   4121  2060.4   13.71 0.000107 ***
Residuals   24   3608   150.3
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

The data argument of the aov() function specifies the location of the variables in the formula. This argument structure is usable in linear modeling as well if any of the variables are part of a larger data frame. Alternatively, it is also possible to attach the cells data frame; in that case there would be no need for adding the data argument.

The output of the summary clearly indicates that genotype has a significant effect on fluorescence ($F=13.71$, $p<0.001$).

As all tests, ANOVA also has assumptions: (1) normal distribution of the residuals and (2) similar variances in all groups (homoscedasticity). These can be assessed visually using diagnostic plots, just like in linear models (the first two plots are of interest again):

```
> par(mfrow=c(1,2)
> plot(anova1)
> par(mfrow=c(1,1)
```



The assumption of homoscedasticity is met and the QQ-plot also seems acceptable.

However, we still do not know which genotype differs from which. To check this, we can use *Tukey's test*, which performs post hoc pair-wise comparisons, while accounting for multiple comparisons. The test is done on the ANOVA object in R as follows:

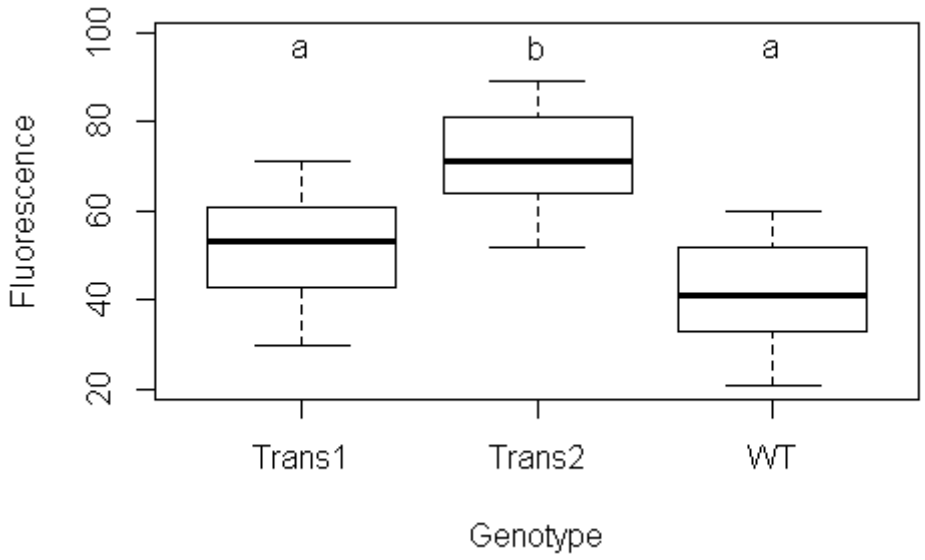
```
> TukeyHSD(anova1)
```

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = Fluorescence ~ Genotype, data = cells)

$Genotype
                    diff        lwr        upr     p adj
Trans2-Trans1  19.55556   5.121895  33.989216 0.0066852
WT-Trans1     -10.22222 -24.655883   4.211438 0.2014297
WT-Trans2     -29.77778 -44.211438 -15.344117 0.0000814
```

The column of adjusted p-values is of interest in the output. It confirms that the transgenic 2 genotype differs significantly from both the wild type ($p<0.001$) and the transgenic 1 ($p=0.007$), while the transgenic 1 does not differ from the wild type ($p=0.201$). Note that no further test statistic is provided in the output of the Tukey's test.

Such results are best illustrated on a figure, preferably on a boxplot, because it gives some hint about the distribution of the data. Statistically confirmed differences between groups are also indicated on these figures. There are several options to do this; e.g. different lowercase letters can indicate differing groups, or asterisks can also be used to indicate differences but even coloring/shading can be used for the purpose. In the plot below, lower case letters are used to identify different groups:

```
> plotTop=max(cells$Fluorescence+10)
> boxplot(Fluorescence~Genotype,data=cells,xlab="Genotype",
+         ylab="Fluorescence",ylim=c(min(cells$Fluorescence),plotTop))
> text(1,plotTop-2,"a")
> text(2,plotTop-2,"b")
> text(3,plotTop-2,"a")
```

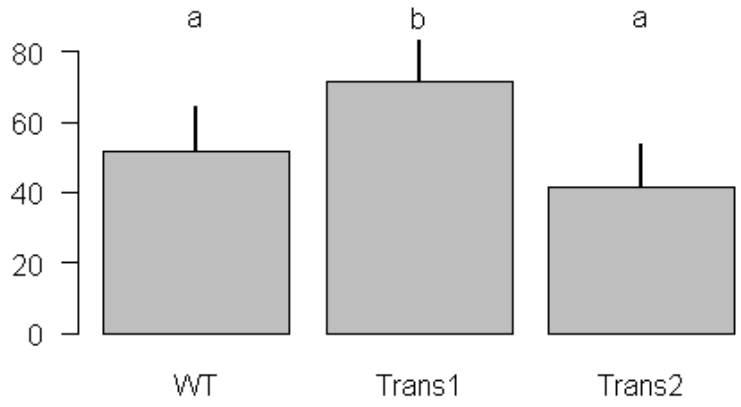


Note that it is necessary to adjust the top of the plot, because otherwise the letters do not have enough space to be shown properly. The + sign at the beginning of the third script line indicates a line break, i.e. the script was broken into two lines in the script window with a

simple Enter for convenience. If having such multiple-line scripts, highlight all lines and then press Ctrl+Enter to run the entire script properly.

An alternative visual representation of the results is the bar chart or bar plot. A bar chart does not give as much detail on the distribution of the data but shows only the mean and some error bars, such as the standard deviation or the standard error of the mean. While standard deviation gives information on the spread of the data, standard error of the mean (SEM) informs about the extent of error in the approximation of the population mean using the sample mean. Standard error is always smaller than standard deviation (and therefore preferred by researchers…) and can be calculated by dividing the standard deviation by the square root of the sample size. If choosing barplot, always use error bars, and specify the type of error bar (SD or SEM) in the figure caption. Drawing barplots is not very straightforward in R. Without using any specified graphic package, like ggplot2, the below scripts represent a possible solution. (Good quality barplots can also be drawn simply with MS Excel.)

```
> means=tapply(cells$Fluorescence,cells$Genotype,mean)
> names=c("WT","Trans1","Trans2")
> SD=tapply(cells$Fluorescence,cells$Genotype,sd)
> plotTop=max(means+SD*2)
> barCenters=barplot(means, names.arg=names, las=1, ylim=c(0,plotTop))
> segments(barCenters, means, barCenters, means+SD, lwd=2)
> text(barCenters[1],90,"a")
> text(barCenters[2],90,"b")
> text(barCenters[3],90,"a")
```



Whiskers are standard deviations in this figure.

If the assumptions of the ANOVA are not met, there is a non-parametric alternative, the Kruskal-Wallis test. Let's test the effect of age group on Fluorescence with this test. But before doing so, notice that Age_group is ordinal but coded as numeric. So first, we need to convert it to a factor:

```
> cells$Age_group=as.factor(cells$Age_group)
> kruskal.test(Fluorescence~Age_group,data=cells)

        Kruskal-Wallis rank sum test

data:  Fluorescence by Age_group
Kruskal-Wallis chi-squared = 10.3135, df = 2, p-value = 0.00576
```

According to the test, age group has a significant effect on fluorescence (Chi^2=10.31, p=0.006).

Unfortunately, there is no non-parametric *post hoc* test that accounts for multiple comparisons, so we need to perform all pairwise comparisons and then manually adjust the p-values for multiple comparisons. There are several adjusting methods; commonly used ones include the Holm's method, the FDR (false discovery rate) method, etc.

Attaching the cells data frame shortens the scripts, so we include this step:

```
> attach(cells)
> w1=wilcox.test(Fluorescence[Age_group==1],Fluorescence[Age_group==2])
> w1

        Wilcoxon rank sum test with continuity correction

data:  Fluorescence[Age_group == 1] and Fluorescence[Age_group == 2]
W = 20.5, p-value = 0.08477
alternative hypothesis: true location shift is not equal to 0

> w2=wilcox.test(Fluorescence[Age_group==1],Fluorescence[Age_group==3])
> w2

        Wilcoxon rank sum test with continuity correction

data:  Fluorescence[Age_group == 1] and Fluorescence[Age_group == 3]
W = 7, p-value = 0.003534
alternative hypothesis: true location shift is not equal to 0

> w3=wilcox.test(Fluorescence[Age_group==2],Fluorescence[Age_group==3])
> w3

        Wilcoxon rank sum test with continuity correction

data:  Fluorescence[Age_group == 2] and Fluorescence[Age_group == 3]
W = 19.5, p-value = 0.06954
alternative hypothesis: true location shift is not equal to 0

> p.adjust(c(w1[3],w2[3],w3[3]),method="fdr")
   p.value    p.value    p.value
0.08476711 0.01060305 0.08476711

> detach(cells)
```

Note that the output of the tests is a list, with the p-value being the third item. The W values to be reported in a publication can remain the original W values; no adjustment is done on them.

Two-way ANOVA

Like in linear modeling, there can be more than one independent variable in ANOVA as well. In *two-way ANOVA* there are two of these, while in multiple-way ANOVA there can be even more; however, there are very few real-life situations when more than two factors are

considered in explaining the variance of measured data. In the `cells` data frame, there are two such factors, which can be used in a two-way ANOVA:

```
> anova2=aov(Fluorescence~Genotype+Age_group,data=cells)
> summary(anova2)
            Df Sum Sq Mean Sq F value   Pr(>F)
Genotype     2   4121  2060.4  111.34 3.10e-12 ***
Age_group    2   3201  1600.3   86.48 3.78e-11 ***
Residuals   22    407    18.5
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

According to the test, both variables have significant effect, but this is no surprise if looking back to the previous analyses. Tukey's pairwise test can be applied in two-way ANOVA as well. The test will do it for us variable by variable:

```
> TukeyHSD(anova2)
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = Fluorescence ~ Genotype + Age_group, data = cells)

$Genotype
                  diff       lwr        upr     p adj
Trans2-Trans1  19.55556  14.46143  24.649679 0.0000000
WT-Trans1     -10.22222 -15.31635  -5.128099 0.0001363
WT-Trans2     -29.77778 -34.87190 -24.683655 0.0000000

$Age_group
        diff       lwr      upr   p adj
2-1 13.00000  7.905877 18.09412 5.5e-06
3-1 26.66667 21.572544 31.76079 0.0e+00
3-2 13.66667  8.572544 18.76079 2.6e-06
```

Two-way ANOVA can also account for and can test the *interaction* of the independent variables. Interaction in this context means that the effect of one independent variable depends on the values of the other independent variable and vice versa. So, e.g. if genotype affects fluorescence only in a specific age group but has no effect in other groups or the effects are opposite in direction in different age groups, we have interaction between the independent variables. Interaction should not be mixed up with multicollinearity.

If we are interested in interactions, we have to use asterisk in the formula between the independent variables and not a plus sign:

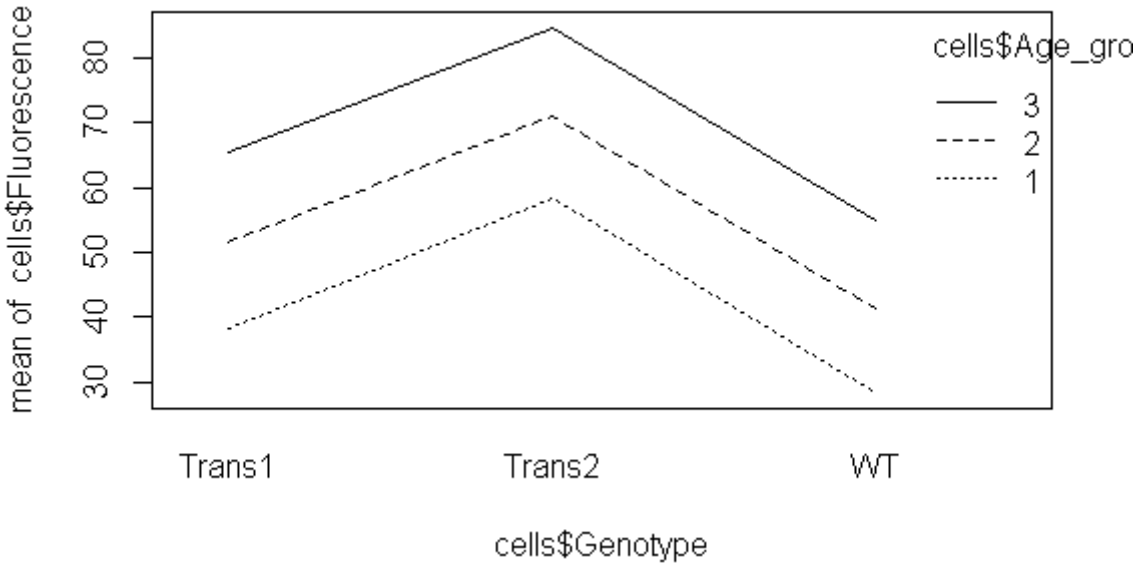
```
> anova3=aov(Fluorescence~Genotype*Age_group,data=cells)
> summary(anova3)
                  Df Sum Sq Mean Sq F value   Pr(>F)
Genotype           2   4121  2060.4  91.200 3.81e-10 ***
Age_group          2   3201  1600.3  70.834 2.94e-09 ***
Genotype:Age_group  4      0     0.1   0.005        1
Residuals          18    407    22.6
---
```

```
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Interaction is indicated with the “Genotyp:Age_group” structure and it seems there is no sign of interaction at all. We do not repeat the *post hoc* test.

The interaction can also be visualized with a simple plot, using the interaction.plot() function:

```
> interaction.plot(cells$Genotype,cells$Age_group,cells$Fluorescence)
```



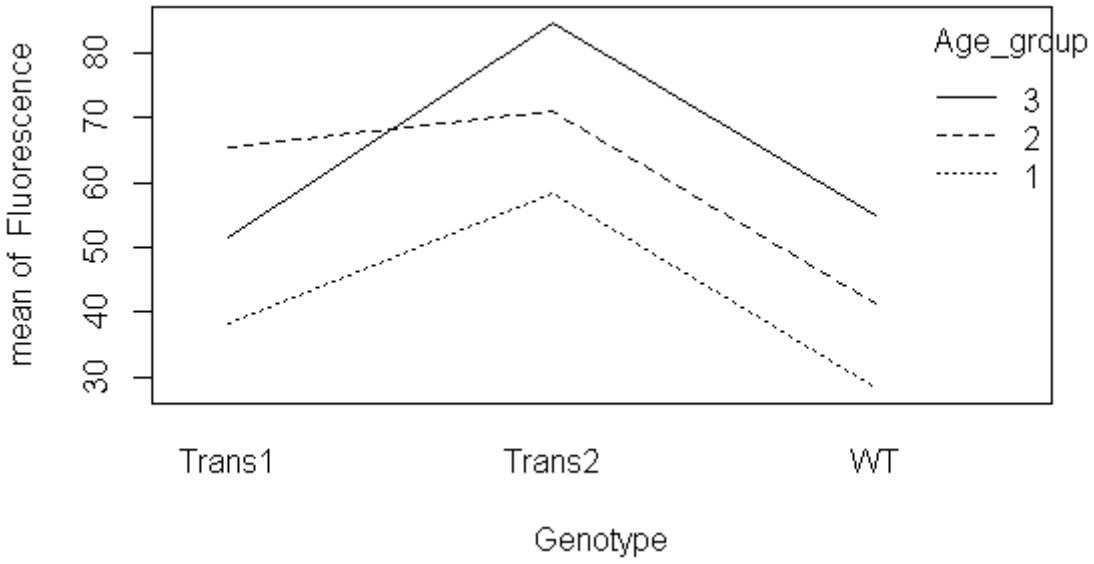
The first two arguments are the independent variables to be checked and the third is the dependent one on which the effects are studied.

Let’s see a modified data set with some, albeit still non-significant interaction. The data9.1.txt is accessible on Coospace.

```
> cells_int=read.table("data9.1.txt",header=T)
> cells_int
   Fluorescence Genotype Age_group
1            21       WT         1
2            31       WT         1
3            33       WT         1
4            40       WT         2
5            41       WT         2
6            43       WT         2
7            52       WT         3
8            53       WT         3
9            60       WT         3
10           30   Trans1         1
11           42   Trans1         1
12           43   Trans1         1
13           49   Trans1         3
14           53   Trans1         3
15           53   Trans1         3
16           61   Trans1         2
17           64   Trans1         2
18           71   Trans1         2
19           52   Trans2         1
20           59   Trans2         1
```

```
21           64   Trans2         1
22           69   Trans2         2
23           71   Trans2         2
24           73   Trans2         2
25           81   Trans2         3
26           84   Trans2         3
27           89   Trans2         3
```

```
> interaction.plot(cells_int$Genotype,cells_int$Age_group,cells_int$Fluorescence)
```




```
> anova4=aov(Fluorescence~Genotype*Age_group,data=cells_int)
> summary(anova4)
                  Df Sum Sq Mean Sq F value   Pr(>F)
Genotype           2   4121  2060.4  35.056 2.03e-07 ***
Age_group          1   2200  2200.1  37.432 4.53e-06 ***
Genotype:Age_group 2    173    86.7   1.475    0.251
Residuals         21   1234    58.8
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

ANOVA can also account for nested design, i.e. a situation when data violate the assumption of independence. The data9.2.txt file also contains fluorescence data but cells were grown on a total of only 10 Petri dishes. Since clones on the same dishes are exposed to the same micro-environmental conditions and these conditions may vary from dish to dish, data are not completely independent, which matches the concept of nested design.

```
> nested=read.table("data9.2.txt",header=T)
> nested
   Cell.type Petri.dish Fluorescence
1          A         T1          158
2          A         T1          156
3          A         T1          160
4          A         T2          139
5          A         T2          142
6          A         T2          135
7          B         T3          185
8          B         T3          180
9          B         T3          184
```

```
10          B          T4          179
11          B          T4          181
12          B          T4          174
13          C          T5          123
14          C          T5          130
15          C          T5          127
16          C          T6          140
17          C          T6          131
18          C          T6          135
19          D          T7          195
20          D          T7          175
21          D          T7          191
22          D          T8          187
23          D          T8          190
24          D          T8          188
25          E          T9          160
26          E          T9          157
27          E          T9          161
28          E         T10          158
29          E         T10          156
30          E         T10          163
```

```
> anova5=aov(Fluorescence~Cell.type/Petri.dish,data=nested)
> summary(anova5)
                    Df Sum Sq Mean Sq F value   Pr(>F)
Cell.type            4  12928    3232 161.059 6.55e-15 ***
Cell.type:Petri.dish 5    714     143   7.113 0.000572 ***
Residuals           20    401      20
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

The output looks like that of anova4 but there is only one real predictor line, Cell.type, which has a significant effect on Fluorescence. Without accounting for the nested design, the test statistic and the p-value are a bit different (and less appropriate).

```
> anova6=aov(Fluorescence~Cell.type,data=nested)
> summary(anova6)
            Df Sum Sq Mean Sq F value   Pr(>F)
Cell.type    4  12928    3232   72.46 2.21e-13 ***
Residuals   25   1115      45
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

SUMMARY

More than two samples → ANOVA

    If significant → Tukey's post hoc test

If assumptions not met → Kruskal-Wallis test

    If significant → pairwise Mann-Whitney + correction for multiple comparisons

More than one nominal predictor → two-way or multiple-way ANOVA

    Interactions are testable

Nested design → Nested ANOVA

Repeated measurements → Repeated measures ANOVA

R functions of Chapter 7

aov

TukeyHSD

barplot

kruskal.test

p.adjust

interaction.plot

Chapter 8

Tests for probabilities and proportions

Parametric tests can be applied only to numeric variables and non-parametric ones to numeric and with some limitations to ordinal variables. However, none are suitable for the statistical evaluation of nominal data. From nominal data, we can prepare tables, containing the occurrences of each level of the variable. From this, it is possible to calculate empirical probabilities of each level or calculate proportions of such probabilities. In this chapter we will focus on a restricted but commonly encountered variant of nominal variables, the binary type. From a binary variable, the empirical proportion of the two outcomes is easily calculated by dividing the occurrences of each outcome with the total number of occurrences. When having this proportion, you can ask whether your empirical proportion equal a hypothetical proportion.

Let's see an example: You flipped a coin and got 34 heads out of 50 tosses and asked whether the coin is biased for heads. The H0 is that the coin is symmetrical and the probability of heads and tails is the same. Since there are two outcomes, the probability of each case should be 0.5, meaning that the hypothetical proportion of heads is 0.5. H1 is the case of a biased coin with different probabilities for heads and tails. Our empirical proportion is 34/50=0.68. So, is the difference from 0.5 caused by random chance? This can be answered with a binomial test:

```
> binom.test(x=34,n=50,p=0.5)

        Exact binomial test

data:  34 and 50
number of successes = 34, number of trials = 50, p-value = 0.01535
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.5330062 0.8047958
sample estimates:
probability of success
                  0.68
```

The x argument is the number of outcomes we counted, n is the total number of observations and p is the hypothetical proportion. According to the outcome (p=0.015), there is significant difference from the hypothetical proportion, so the coin is biased.

Just like in t-test, the empirical data can be compared to other empirical data, i.e. two proportions can be compared whether their difference is real or just caused by random chance.

For example, Donald Trump received 53 votes out of 100 votes in village A and received 41 out of 100 in village B. Is there a statistically confirmable difference in the popularity of Trump in the two towns? The answer is calculated with a chi-squared test (also spelled as chi^2 test or χ^2 test).

```
> town1=c(53,47)
> town2=c(41,59)
> votes=cbind(town1,town2) ##chisq.test() needs input data as a matrix
```

```
> rownames(votes)=c("Yes","No")
> votes
    town1 town2
Yes    53    41
No     47    59

> chisq.test(votes)

      Pearson's Chi-squared test with Yates' continuity correction

data:  votes
X-squared = 2.4287, df = 1, p-value = 0.1191
```

The chisq.test() function needs data in the form of a matrix. Be careful to have locations/objects/etc. in columns and the occurrences of each outcome in rows. It is not necessary to name the outcomes.

According to the chi-squared test, there is no difference in the popularity of Trump in the studied towns (chi-squared=2.43, p=0.119).

The calculation of the chi-squared test can encounter difficulties if any record in the input matrix is lower than 5. In such cases, the p-values may need to be simulated to get reliable results. Alternatively, Fisher's exact test is also frequently used.

```
> town3=c(1,98)
> town4=c(10,120)
> votes2=cbind(town3,town4)
> chisq.test(votes2)

      Pearson's Chi-squared test with Yates' continuity correction

data:  votes2
X-squared = 4.1239, df = 1, p-value = 0.04228

Warning message:
In chisq.test(votes2) : Chi-squared approximation may be incorrect
```

There is a warning message with the new data telling us that the results may not be reliable, as town 3 had only one yes vote for Trump. Note also that the total number of counts is not the same in town 3 and 4. Equal number of counts is not an assumption of prerequisite for the chi-squared test.

By setting the simulate.p.value argument to true, the p-value is recalculated:

```
> chisq.test(votes2,simulate.p.value=T)

      Pearson's Chi-squared test with simulated p-value (based on 2000
      replicates)

data:  votes2
X-squared = 5.4879, df = NA, p-value = 0.02599
```

The new chi-squared value and p-value are more reliable.

Fisher's exact test yields similar results:

```
> fisher.test(votes2)

      Fisher's Exact Test for Count Data
```

```
data:  votes2
p-value = 0.02577
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.002798097 0.893028110
sample estimates:
odds ratio
 0.1232511
```

SUMMARY

Empirical probability or proportion vs. hypothetical one → binomial test

Two empirical probabilities or proportions → chi-squared test

If any count < 5 → simulate p-values or Fisher's exact test

R functions of Chapter 8

binom.test

chisq.test

fisher.test

Chapter 9

Survival analysis

Several biological studies are designed to follow up the survival of subjects after a treatment or other impacts the subjects are exposed to. These subjects can be individual cells, lab animals, patients, endangered wild animals, etc. The main focus of survival analysis is on the rate of survival over time. This basically means that the f(t)=p(S) survival function (the probability of being alive at time t) is to be assessed.

In these studies, the investigator starts with a certain set of subjects and later on checks how many are still alive. Enrollment of new subjects during the study period is also possible, as well as the loss of subjects due to unknown reasons or known reasons that are different from the scrutinized effect. In survival analysis, those patients that become unavailable for further check-up in these ways are frequently called *censored* subjects.

The situation is better illustrated by a real data set, such as the `melanom` data frame of the ISwR package. To access the data, you need to install and load the package in the Environment.

```
> head(melanom)
   no status days ulc thick sex
1 789      3   10   1   676   2
2  13      3   30   2    65   2
3  97      2   35   2   134   2
4  16      3   99   2   290   1
5  21      1  185   1  1208   2
6 469      1  204   1   484   2

> str(melanom)
'data.frame':  205 obs. of  6 variables:
 $ no    : int  789 13 97 16 21 469 685 7 932 944 ...
 $ status: int  3 3 2 3 1 1 1 1 3 1 ...
 $ days  : int  10 30 35 99 185 204 210 232 232 279 ...
 $ ulc   : int  1 2 2 2 1 1 1 1 1 1 ...
 $ thick : int  676 65 134 290 1208 484 516 1288 322 741 ...
 $ sex   : int  2 2 2 1 2 2 2 2 1 1 ...
```

The melanoma data set contains the data of patients with melanoma malignum (a type of cancer), who received a certain surgical treatment. Rows correspond to patients: 205 in total as indicated by the structure call. The first column ("no") is the patient ID and the second column ("status") tells the fate of the patient. Status 1 means that the patient died of melanoma x days after the treatment but before the study was terminated. Status 2 means that the patient was still alive at the end of the study; corresponding days vary due to different times of enrollment in the study (time of surgery). Patients with status 3 died x days after the surgery but due to reasons unrelated to their melanoma. The fourth column ("ulc") tells whether the tumor was ulcerated at the time of treatment (1 = yes, 2 = no). The "thick" column is thickness of the tumor and "sex" is the gender of the patient (1 = female, 2 = male).

First, we should make a copy of the original data frame by storing it under a different name:

```
> melanom1=melanom
```

Status 2 and 3 patients are essentially identical for survival analysis; they became unavailable for further follow-up. These patients are the censored ones according to the terminology. To ease further analysis, we should change their status to 0:

```
> melanom1[melanom1$status==2,2]=0
> melanom1[melanom1$status==3,2]=0

> str(melanom1)
'data.frame':  205 obs. of  6 variables:
 $ no    : int  789 13 97 16 21 469 685 7 932 944 ...
 $ status: num  0 0 0 0 1 1 1 1 0 1 ...
 $ days  : int  10 30 35 99 185 204 210 232 232 279 ...
 $ ulc   : int  1 2 2 2 1 1 1 1 1 1 ...
 $ thick : int  676 65 134 290 1208 484 516 1288 322 741 ...
 $ sex   : int  2 2 2 1 2 2 2 2 1 1 ...
```

The new melanom1 data frame has only 0 and 1 in the status column; patients with 1 died due to the cause of interest, while patients with 0 had a censoring event of any kind (died of other causes or the study ended while still alive) after as many days of follow-up as indicated in the "days" column.

Survival analysis is done with the survival package, so it has to be installed and loaded in. Once having access to the functions of the package, the next step is to create a survival object from the melanom1 data frame.

```
> s=Surv(melanom1$days,melanom1$status)
> s
[1]   10+   30+   35+   99+  185   204   210   232   232+  279   295   355+
386   426   469   493+  529   621   629
[20]  659   667   718   752   779   793   817   826+  833   858   869   872
967   977   982  1041  1055  1062  1075
[39] 1156  1228  1252  1271  1312  1427+ 1435  1499+ 1506  1508+ 1510+ 1512
+ 1516  1525+ 1542+ 1548  1557+ 1560  1563+
[58] 1584  1605+ 1621  1627+ 1634+ 1641+ 1641+ 1648+ 1652+ 1654+ 1654+ 1667
1678+ 1685+ 1690  1710+ 1710+ 1726  1745+
[77] 1762+ 1779+ 1787+ 1787+ 1793+ 1804+ 1812+ 1836+ 1839+ 1839+ 1854+ 1856
+ 1860+ 1864+ 1899+ 1914+ 1919+ 1920+ 1927+
[96] 1933  1942+ 1955+ 1956+ 1958+ 1963+ 1970+ 2005+ 2007+ 2011+ 2024+ 2028
+ 2038+ 2056+ 2059+ 2061  2062  2075+ 2085+
[115] 2102+ 2103  2104+ 2108  2112+ 2150+ 2156+ 2165+ 2209+ 2227+ 2227+ 225
6  2264+ 2339+ 2361+ 2387+ 2388  2403+ 2426+
[134] 2426+ 2431+ 2460+ 2467  2492+ 2493+ 2521+ 2542+ 2559+ 2565  2570+ 266
0+ 2666+ 2676+ 2738+ 2782  2787+ 2984+ 3032+
[153] 3040+ 3042  3067+ 3079+ 3101+ 3144+ 3152+ 3154+ 3180+ 3182+ 3185+ 319
9+ 3228+ 3229+ 3278+ 3297+ 3328+ 3330+ 3338
[172] 3383+ 3384+ 3385+ 3388+ 3402+ 3441+ 3458+ 3459+ 3459+ 3476+ 3523+ 366
7+ 3695+ 3695+ 3776+ 3776+ 3830+ 3856+ 3872+
[191] 3909+ 3968+ 4001+ 4103+ 4119+ 4124+ 4207+ 4310+ 4390+ 4479+ 4492+ 466
8+ 4688+ 4926+ 5565+
```

A survival object is not a statistical term, it is only a predefined data structure that can be handled with the functions of the survival package. The Surv() function that creates it has two

compulsory arguments. The duration until an event is provided first (days in this case) and the second is the type of the event (see above for options). The + sign in the survival object indicates censoring event. The head() and tail() functions also work on survival objects, so it is not necessary to print all records in it to have a brief look. Also, the survival object appears in the upper right window as

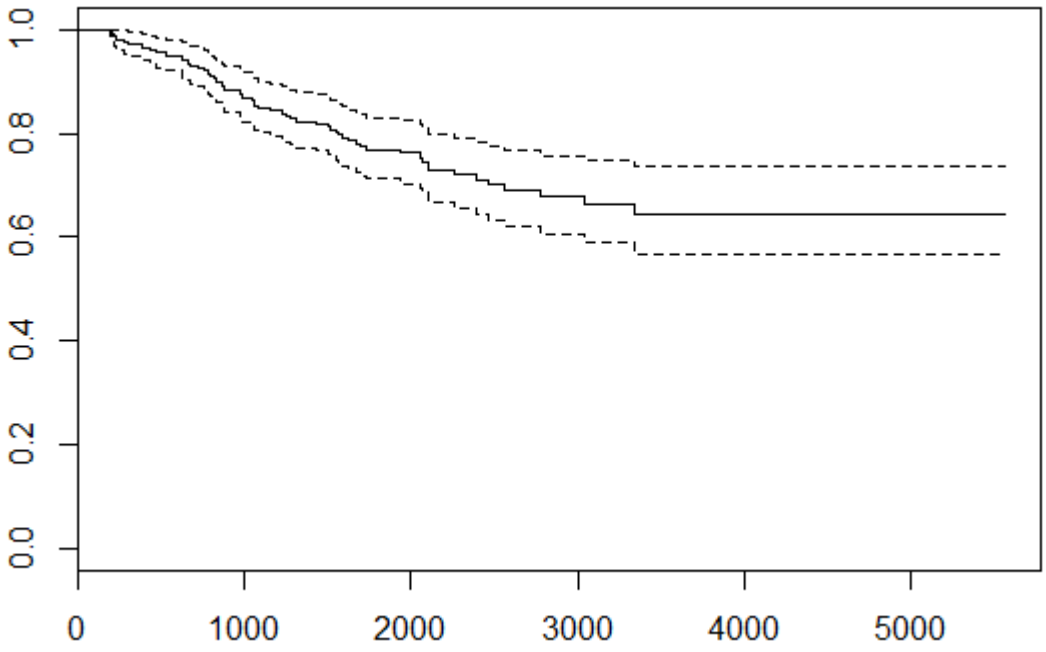
```
s      | Surv [1:205, 1:2] 10+ 30+ 35+ 99+ 185 204 210 …
```

The survfit() function is designed to process a survival object as follows:

```
> s1=survfit(s~1)
> s1
Call: survfit(formula = s ~ 1)

      n  events  median 0.95LCL 0.95UCL
    205      57      NA      NA      NA

> summary(s1)
Call: survfit(formula = s ~ 1)

 time n.risk n.event survival std.err lower 95% CI upper 95% CI
  185    201       1    0.995 0.00496        0.985        1.000
  204    200       1    0.990 0.00700        0.976        1.000
  210    199       1    0.985 0.00855        0.968        1.000
  232    198       1    0.980 0.00985        0.961        1.000
  279    196       1    0.975 0.01100        0.954        0.997
  295    195       1    0.970 0.01202        0.947        0.994
  386    193       1    0.965 0.01297        0.940        0.991
  426    192       1    0.960 0.01384        0.933        0.988
  469    191       1    0.955 0.01465        0.927        0.984
  ...    ...     ...    ...   ...            ...          ...
 2108     88       1    0.729 0.03358        0.666        0.798
 2256     80       1    0.720 0.03438        0.656        0.791
 2388     75       1    0.710 0.03523        0.645        0.783
 2467     69       1    0.700 0.03619        0.633        0.775
 2565     63       1    0.689 0.03729        0.620        0.766
 2782     57       1    0.677 0.03854        0.605        0.757
 3042     52       1    0.664 0.03994        0.590        0.747
 3338     35       1    0.645 0.04307        0.566        0.735
```

The survfit function can process a formula, where 1 is provided again to indicate that it is for the event of interest. Simply calling the output is not very informative; it displays that we had 205 subjects, out of which 57 had the event of interest. The median time of survival is NA because more than half of the patients were still alive at the end of the study. The summary of `s1` is more interesting. The first column is the time until an event and the survival in the fourth column tells the estimated probability of being alive after that time. A standard error and lower and upper 95% confidence limits are also given for this estimate. This estimate is called the *Kaplan-Meyer estimate of survival.* The `s1` estimate can be plotted easily to display the survival function:

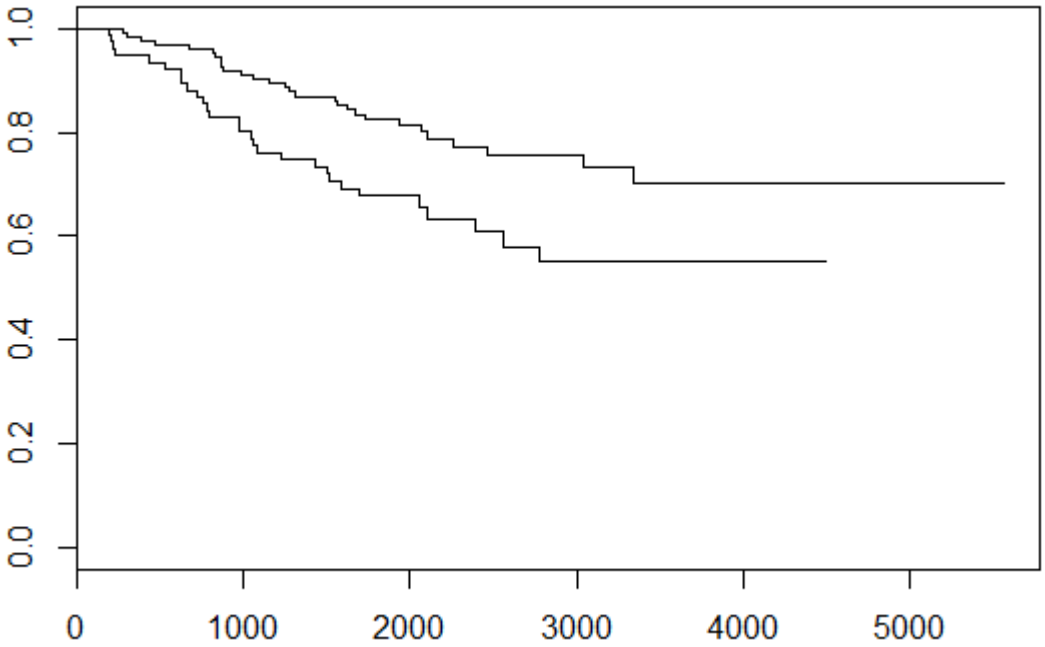
```
> plot(s1)
```



This plot shows the probability of survival over time. Since this function is calculated from a sample, it can differ from the true function of the entire population (patients with melanoma having the same treatment ever) due to random chance. The dashed lower and upper confidence limits confine this true function with a confidence of 95%.

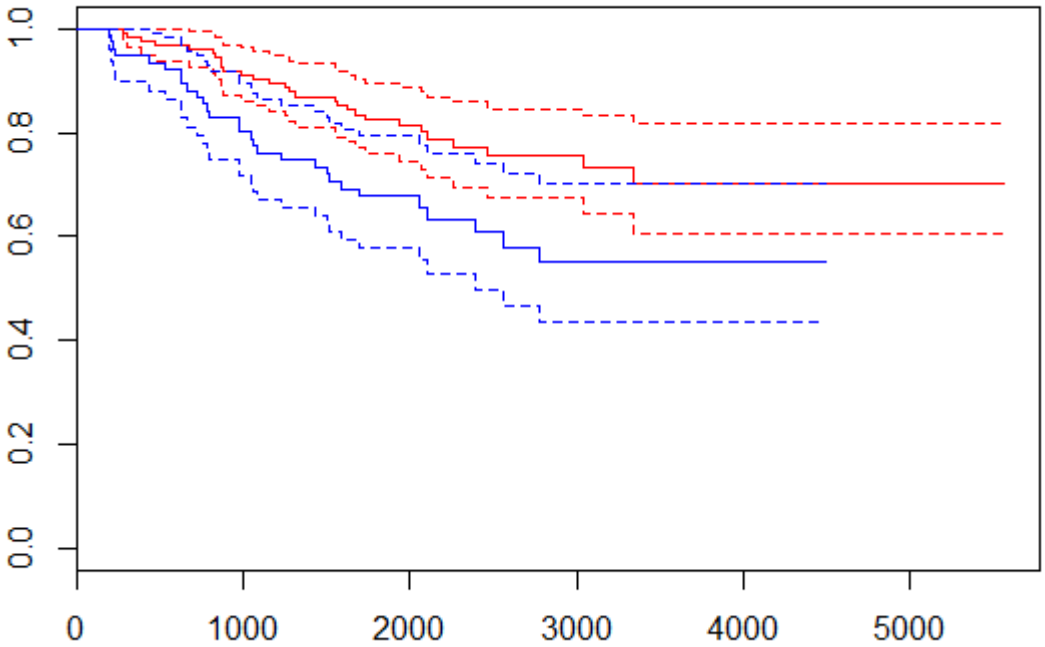
Preparing the survival function is informative on its own, but in studies it is more frequent to compare survival functions of different groups, like the to test the efficiency of different treatments or the efficiency of the same treatments on different subject groups. Here we have female and male patients, so we can compare whether the survival rate depends on gender. Different groups can be compared also with the survfit() function, but the survival object has to be modified a bit for this:

```
> s2=Surv(melanom1$days,melanom$status==1)
> surv.by.gender=survfit(s2~melanom1$sex)
> plot(surv.by.gender)
```



As a default setting, confidence limits are turned off when plotting more than one survival function in the same plot to avoid overlapping curves. It possible to turn the confidence limits on and to use colors to indicate which gender is which on the plot.

```
> plot(surv.by.gender,conf.int=T,col=c("blue","red"))
```



Red is the color for the gender coded with 1 (females) and blue is for gender 2 (males).

As you can see, the survival function of males is below that of females, indicating that females have higher survival rate. However, the confidence bands overlap a bit, so we cannot

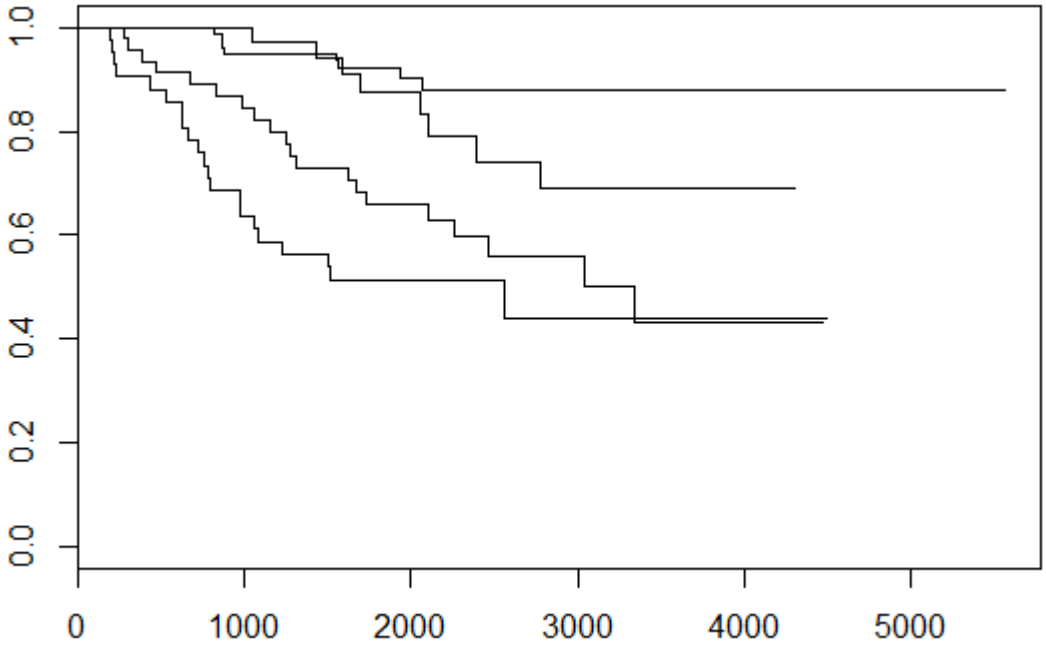
tell if the difference is caused by random chance or it is an acceptable difference. The difference can be tested for significance with a log-rank test, using the survdiff() function:

```
> survdiff(s2~melanom1$sex)
Call:
survdiff(formula = s2 ~ melanom1$sex)

                 N Observed Expected (O-E)^2/E (O-E)^2/V
melanom1$sex=1 126       28     37.1      2.25      6.47
melanom1$sex=2  79       29     19.9      4.21      6.47

 Chisq= 6.5  on 1 degrees of freedom, p= 0.011
```

The test calculates a chi-squared test statistic and gives a p-value. As it is low enough we can accept the difference and report our findings like this: Females have a statistically higher survival rate than males if treated with the studied surgical technique (chi-squared=6.5, p=0.011).

Multiple groupings of survival records, i.e. the use of more than one predictor is not advisable as the test cannot calculate the effect of separate predictors but will combine them. In practice this means if we include another nominal (or numeric) variable in the formula in the survdiff() function, the levels would be combined and each combination would be handled as a separate group. The output will contain the test of the difference between the combined groups. The same applies for the Kaplan-Meyer estimate calculated this way and plotted on a graph as follows:

```
> survdiff(s2~melanom1$sex+melanom1$ulc)
Call:
survdiff(formula = s2 ~ melanom1$sex + melanom1$ulc)

                                 N Observed Expected (O-E)^2/E (O-E)^2/V
melanom1$sex=1, melanom1$ulc=1 47       20    12.44     4.602      5.89
melanom1$sex=1, melanom1$ulc=2 79        8    24.71    11.298     19.98
melanom1$sex=2, melanom1$ulc=1 43       21     8.77    17.047     20.26
melanom1$sex=2, melanom1$ulc=2 36        8    11.09     0.859      1.07

 Chisq= 34  on 3 degrees of freedom, p= 1.98e-07


> surv.by.gender2=survfit(s2~melanom1$sex+melanom1$ulc)
> plot(surv.by.gender2)
```



The survival functions in the plot correspond to combined groups, like females with ulceration, females with no ulceration, etc.

If it is needed to test the effect of more than one predictor on survival in a single model, the *Cox proportional hazards model* (or simply Cox regression) can be recommended. Cox regression is calculated also with the functions of the survival package. When including only gender, the formulation and the output are as follows:

```
> cox=coxph(s2~melanom1$sex)
> summary(cox)
Call:
coxph(formula = s2 ~ melanom1$sex)

  n= 205, number of events= 57

                coef exp(coef) se(coef)     z Pr(>|z|)
melanom1$sex 0.6622    1.9390   0.2651 2.498   0.0125 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

             exp(coef) exp(-coef) lower .95 upper .95
melanom1$sex     1.939     0.5157     1.153      3.26

Concordance= 0.59  (se = 0.033 )
Rsquare= 0.03   (max possible= 0.937 )
Likelihood ratio test= 6.15  on 1 df,   p=0.01314
Wald test            = 6.24  on 1 df,   p=0.01251
Score (logrank) test = 6.47  on 1 df,   p=0.01098
```

Cox regression works with hazards ratios, which equal the ratio of the death rates of the groups to be compared. In practice, this means that if males have a death rate twice as high as females, the hazard ratio will be 2 (or 0.5 if viewed from the opposite direction). The coef value in the output is the logarithm of this hazard ratio, while the exp(coef) corresponds to the

exact hazard ratio. The inverted ratio is also provided in the output below the significance codes. The hazard ratio is tested for significance and yielded a z-value of 2.50 and a p-value of 0.012, so the Cox regression also confirmed that gender significantly affects survival.

The entire model is also tested for significance using three different methods; their results are also significant.

Like in multiple linear regression or the two-way ANOVA, it is allowed to include more predictors:

```
> cox2=coxph(s2~melanom1$sex+melanom$ulc)
> summary(cox2)
Call:
coxph(formula = s2 ~ melanom1$sex + melanom$ulc)

  n= 205, number of events= 57

                coef exp(coef) se(coef)      z Pr(>|z|)
melanom1$sex  0.5165    1.6761   0.2667  1.937   0.0528 .
melanom$ulc  -1.4180    0.2422   0.2969 -4.775 1.79e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

             exp(coef) exp(-coef) lower .95 upper .95
melanom1$sex    1.6761     0.5966    0.9938    2.8268
melanom$ulc     0.2422     4.1289    0.1353    0.4334

Concordance= 0.719  (se = 0.038 )
Rsquare= 0.145   (max possible= 0.937 )
Likelihood ratio test= 32.16  on 2 df,   p=1.039e-07
Wald test            = 28.59  on 2 df,   p=6.206e-07
Score (logrank) test = 33.51  on 2 df,   p=5.277e-08
```

According to this output, if gender and ulceration are modelled together, the significant effect of gender is lost for the effect of ulceration. the full model is again confirmed to be significant by every model at the end of the output.

SUMMARY

Survival analysis: estimating the survival function

Kaplan-Meier estimate: calculated for individual groups

Comparing groups:

- Log-rank test
- Cox proportional hazards model

R functions in Chapter 9

Surv

survfit

survdiff

coxph

Chapter 10

Multivariate statistics

Multivariate data arise when we measure several variables on each sampling unit. In most cases the variables are related in such a way that all variables must be analysed simultaneously. In order to summarize the multivariate data we need summary statistics for each variable separately (e.g. means and variances) and summary for their relationships (e.g. covariances and correlations). Furthermore, the concept of distance between observations is used in many methods. There are also many graphical procedures to visualise the pattern. There are also multiviariate analogues of some single variate statistical test (for example t-test or ANOVA). But the aims of the multivariate statistical methods are often different from the others mentioned in the previous chapters. There are approaches that assume a given structure of the data and divide cases into groups while others seek directly the structure (groups, relationships). The choice among these exploratory approaches and specific methods depends on the question that we aim to answer. The following methods will be outlined:

- principal components analysis (`prcomp`)
- multidimensional scaling (`cmdscale` and `isoMDS`)
- factor analysis (`factanal`)
- cluster analysis (`hclust`, `kmeans`) and recursive partitioning (`tree`)
- discriminant analysis (`lda`)

Some of these techniques are available as part of the basic installation (stats package), while others are implemented in the MASS package and there are many specialized packages for a given set of problems (see CRAN Multivariate task view for an overview). As an example we use the vegan package in ecological and evolutionary studies. Short introduction to multivariate analysis using R is given in Crawley (2012) or Everitt & Hothorn (2011).

Data are generally given in a data frame (`data.frame`) that is imported from a text file (e.g. with functions `read.table` or `read.csv`). We will use the following datasets:

- "**plantdw**": The "plantdw.txt" file (subset from "pgfull.txt", Crawley 2012) contains mean dry weights for 31 plant species (columns from 1 to 31) on 50 plots (rows). It contains also information about the plots ("plot" and "plotclass" columns) and the covariates (hay biomass and soil pH).
- "**taxon**": The "planttax.txt" file contains measurements of 5 variables on 80 individual plants (subset from "taxon.txt", Crawley 2012). This is an artificial example; we know that the first 20 rows come from group 1, rows 21-40 from group 2, rows 41-60 from group 3 and rows 61-80 from group 4.

Most of the following analyses are reproduced from Crawley 2012, more details are given in the book.

Data

First we read the data and create a new data frame ("d") that contains the 50 plots and the first 31 columns of the original data set.

```
> d.full=read.table("plantdw.txt",header=T)
> names(d.full)
 [1] "AC"        "AE"        "AM"        "AO"        "AP"        "AS"
 [7] "BH"        "BM"        "CM"        "CN"        "DG"        "FR"
[13] "HL"        "HP"        "HS"        "KA"        "LC"        "LH"
[19] "LM"        "LO"        "LP"        "PL"        "PP"        "PT"
[25] "RA"        "RC"        "SM"        "TF"        "TG"        "TO"
[31] "TP"        "plot"      "plotclass" "biomass"   "pH"
> d=d.full[,1:31]
> str(d)
'data.frame':   50 obs. of  31 variables:
 $ AC: num  11.3 12.4 4.84 7.06 1.98 ...
 $ AE: num  0.34 0.14 0 0.15 6.61 6.4 0.22 0.23 5.29 0.32 ...
...
 $ TP: num  0.33 0.01 0.68 0.16 9.15 2.17 0.03 1.21 1.26 0 ...
```

Next, we create new labels (denoted by "l") to display together the treatment ("plotclass" coded by letters) with the plot identifier.

```
> table(letters[d.full$plotclass])
 a  b  c  d
13 15 13  9
> l = paste(d.full$plot,letters[d.full$plotclass],sep=""); l
 [1] "10c"   "3d"    "2.1a"  "12c"   "15b"   "19.3b" "2.1c"
 [8] "9.1c"  "19.1c" "17a"   "17c"   "6b"    "1c"    "11.1a"
[15] "13.2c" "11.1b" "7c"    "18.1a" "3b"    "11.2c" "11.2a"
[22] "16a"   "18.2a" "4.2d"  "12a"   "14.1a" "8b"    "14.1b"
[29] "7a"    "18.1b" "20.3b" "8a"    "8c"    "7b"    "7d"
[36] "8d"    "4.1c"  "4.2b"  "13.2b" "12d"   "15d"   "14.1c"
[43] "11.1d" "6a"    "17b"   "14.2d" "3a"    "1b"    "2.1d"
[50] "9.1b"
```

Distances

Similarity of two objects on the basis of the variables is a fundamental porperty in many methods. Similarity is deduced from multivariate distance, the measure of the dissimilarity. Distance of the objects is stored in a distance matrix, which can be created by the `dist` function. Several different distances can be used, the default is the Euclidean distance (see `?dist`).

```
> dist(d)
           1         2         3         4         5         6         ...
2  16.505681
3  21.910068 10.176473
4  14.523908  6.662282  8.281715
5  25.977679 17.823987 14.435162 16.471032
```

```
6  25.510257 12.948046 15.097043 16.478759 14.980841
7  15.405136  3.726298  9.150164  4.717913 17.029486 14.692556
...
49 16.644014  4.028039  9.278960  6.112896 17.015928 14.446391         ...
50 26.537688 17.276921 14.565143 16.631939 13.664981 15.510000         ...
```

Small distance (e.g. approximately 16.5 between object 1 and 2) means large similarity.

Principal components analysis

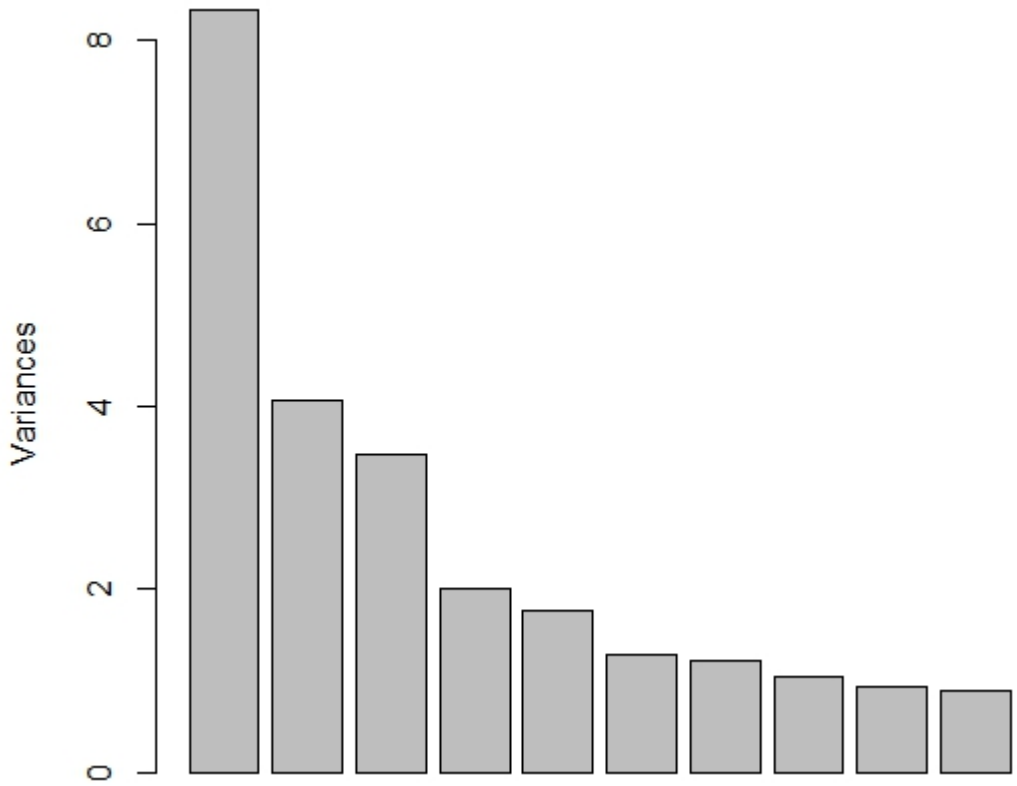
Principal components analysis (PCA) tries to find linear combinations of the variables that capture most of the variation in the data. The aim is to consider a small number of combinations of variables, called principal components (PC), associated with the objects. Principal components are uncorrelated and ordered so that the first few account for most of the variation in all the original variables. We analyse further the "plantdw" dataset.

PCA is carried out by the method using `prcomp` with the option scale=TRUE, because the variances are different for the plant species (check it, e.g. `apply(d,2,var)`). Note that there is another approach for PCA implemented in the stats package (see `?princomp`).

```
> m=prcomp(d, scale=T)
> summary(m)
Importance of components:
                          PC1    PC2    PC3    PC4     PC5     PC6     PC7
Standard deviation     2.8855 2.0150 1.8666 1.4173 1.33071 1.12948 1.10706
Proportion of Variance 0.2686 0.1310 0.1124 0.0648 0.05712 0.04115 0.03954
Cumulative Proportion  0.2686 0.3996 0.5120 0.5767 0.63387 0.67502 0.71456
                           PC8     PC9    PC10    PC11    PC12   PC13    PC14
Standard deviation     1.02757 0.96469 0.94519 0.93127 0.88012 0.8239 0.77330
Proportion of Variance 0.03406 0.03002 0.02882 0.02798 0.02499 0.0219 0.01929
Cumulative Proportion  0.74862 0.77864 0.80746 0.83543 0.86042 0.8823 0.90161
                          PC15   PC16    PC17    PC18    PC19    PC20    PC21
Standard deviation     0.69112 0.6865 0.60113 0.57626 0.53241 0.48481 0.39985
Proportion of Variance 0.01541 0.0152 0.01166 0.01071 0.00914 0.00758 0.00516
Cumulative Proportion  0.91702 0.9322 0.94387 0.95459 0.96373 0.97131 0.97647
                          PC22    PC23    PC24    PC25    PC26    PC27    PC28
Standard deviation     0.3818 0.36704 0.33707 0.33126 0.26256 0.24523 0.19391
Proportion of Variance 0.0047 0.00435 0.00367 0.00354 0.00222 0.00194 0.00121
Cumulative Proportion  0.9812 0.98552 0.98918 0.99272 0.99495 0.99689 0.99810
                          PC29   PC30    PC31
Standard deviation     0.17927 0.1359 0.09117
Proportion of Variance 0.00104 0.0006 0.00027
Cumulative Proportion  0.99914 0.9997 1.00000
```

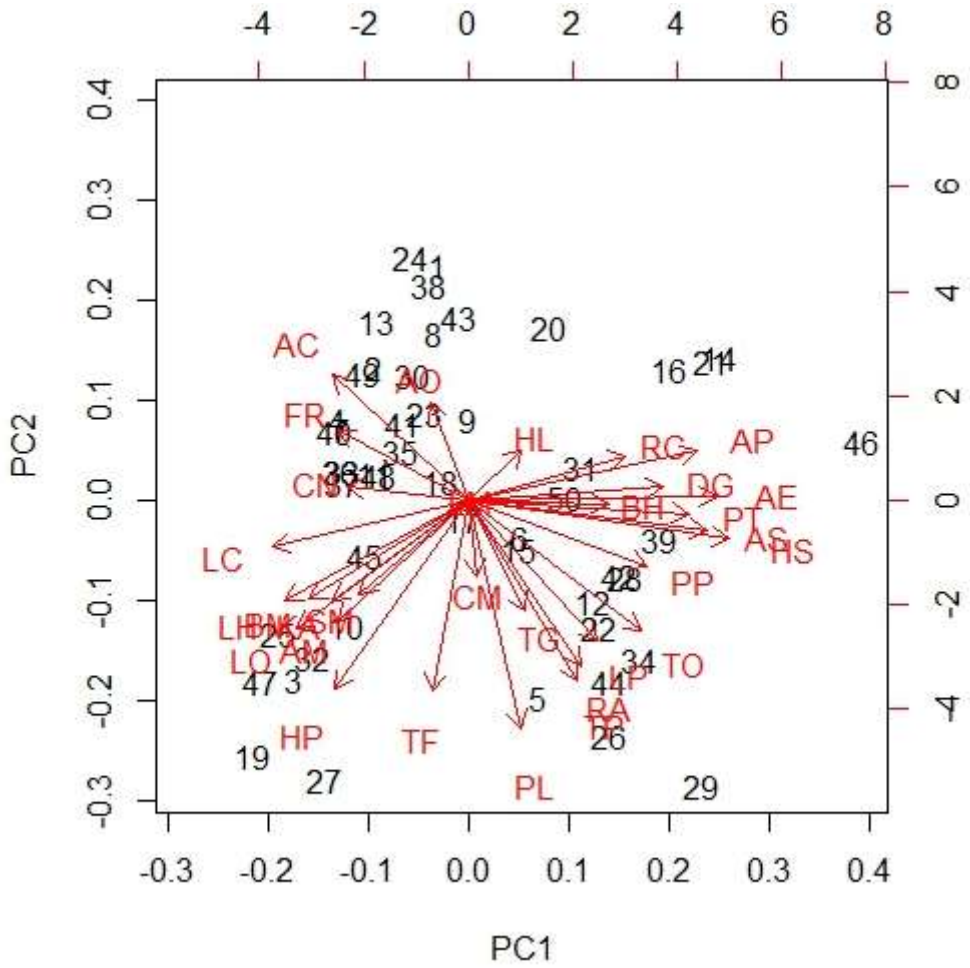
The first principal component explains 26.9% of the total variation; the second explains 13.1% etc. PC1-PC7 together accounts for almost 72% of the total variation. We need 14 components to achieve 90%. The plot of the model (called scree plot) shows the relative importance of each principal component.

```
> plot(m, main="")
```



Principal component loadings, defined as the influence of the original variables on the principal components, can be displayed by using `biplot`. Loadings can be extracted using `m$rotation` (it is an element of the result list of `prcomp`).

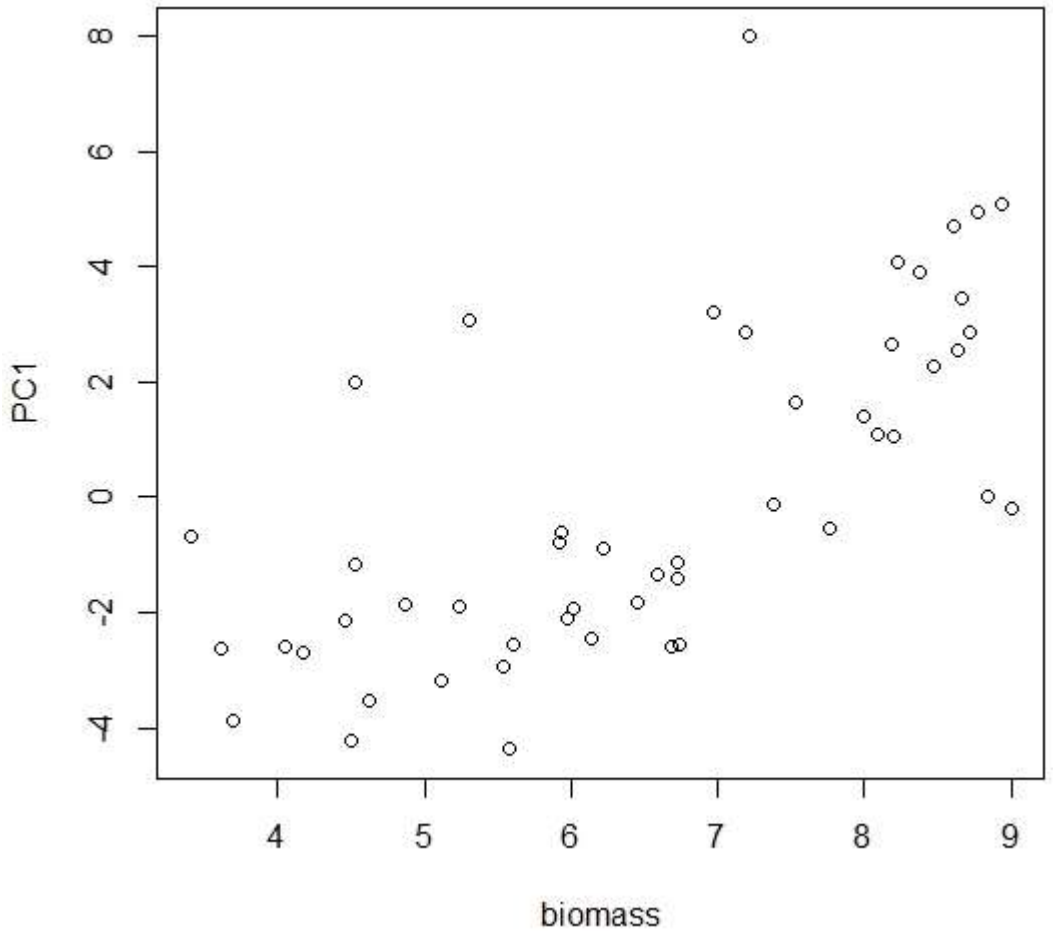
```
> biplot(m, main="")
```



The numbers represent the rows of the original data frame (the plots) and the arrows show the relative loadings of the species (the original variables) on the first and second principal components. Species AP, AE and HS have strong positive loadings on PC1 and LC and LH have strong negative loadings (among others).

Explanatory variables can be plotted against the principal components to look for patterns. Principal components of the original plots can be extracted using `predict`. PC1 is the first column of matrix returned by the function (see `predict(m)[1:5,1:5]; predict(m)[,1]`).

```
> plot(d.full$biomass, predict(m)[,1], xlab="biomass", ylab="PC1")
```



This relationship suggests that the first principal component is associated with the increasing biomass (from Crawley 2012).

PCA considers relationships within a set of variables. Canonical correlation analysis (see e.g. `cancor` in R stats package) is used widely for insights into the association between two sets of variables. Correspondence analysis is used in the same way as PCA but for categorical variables (see e.g. `corresp` in MASS package). Reducing dimesionality of the original data and ordering objects according to their similarity (called ordination or gradient analysis) is addressed by many different methods.

Exercises

1. Show the effect of 14 components in the scree plot!
2. What kind of species has strong positive and negative loadings on PC2?
3. What do you think about the association of PC2 and pH? Plot the relationship!

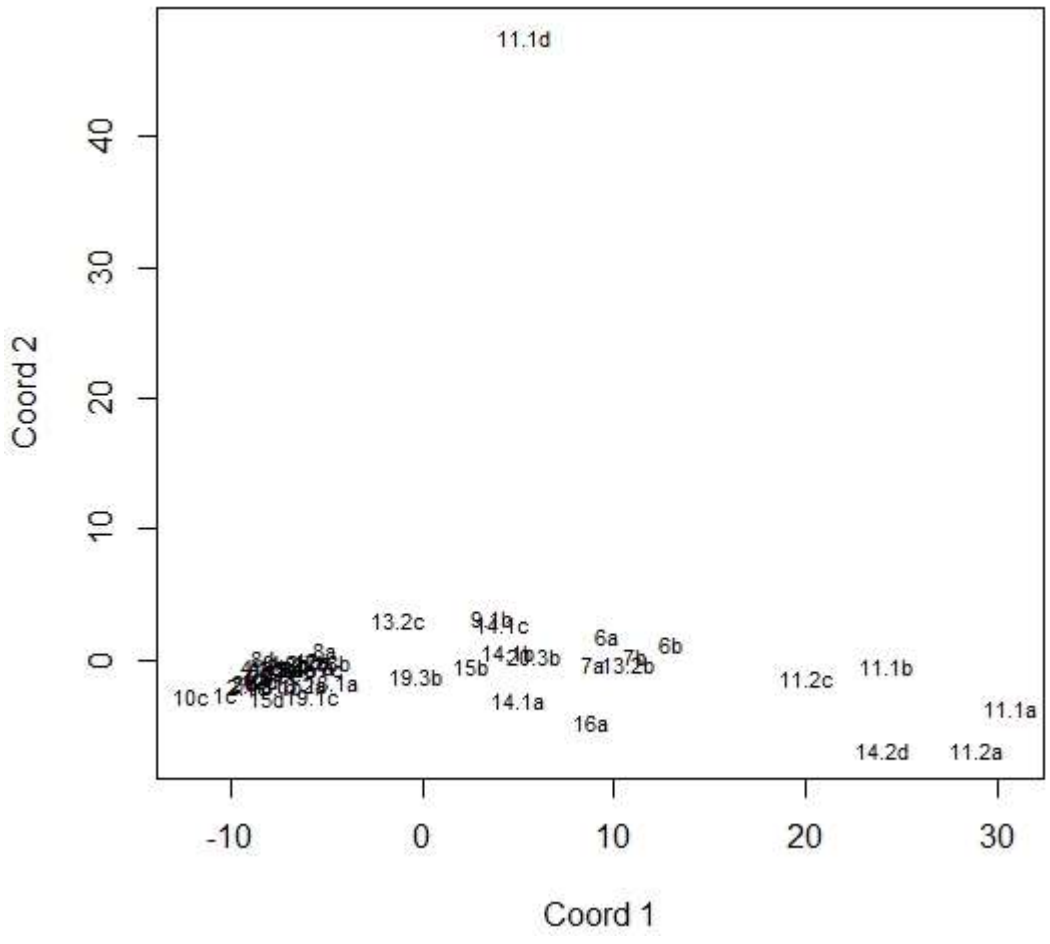
Multidimensional scaling

Metric multidimensional scaling (also known as principal coordinates analysis, PCoA) is conceptually very similar to PCA; both preserve the Euclidean distances as far as possible.

But multidimensional scaling is applied to the distances, which are derived from the data. This provides more flexibility for this method. We carry out PCoA using `cmdscale`. The maximum dimensions must be specified (default is two) in this implementation. The scores of the objects are obtained in a numeric matrix (by default). We use now the the new labels for plotting.

```
> m = cmdscale(dist(d))
> m
              [,1]        [,2]
 [1,] -12.1099520 -2.69112871
 [2,]  -9.2996918 -1.73565334
 [3,]  -6.5830315 -0.53596952
...
[49,]  -8.8591645 -1.79471985
[50,]   3.7434723  3.19985487
```

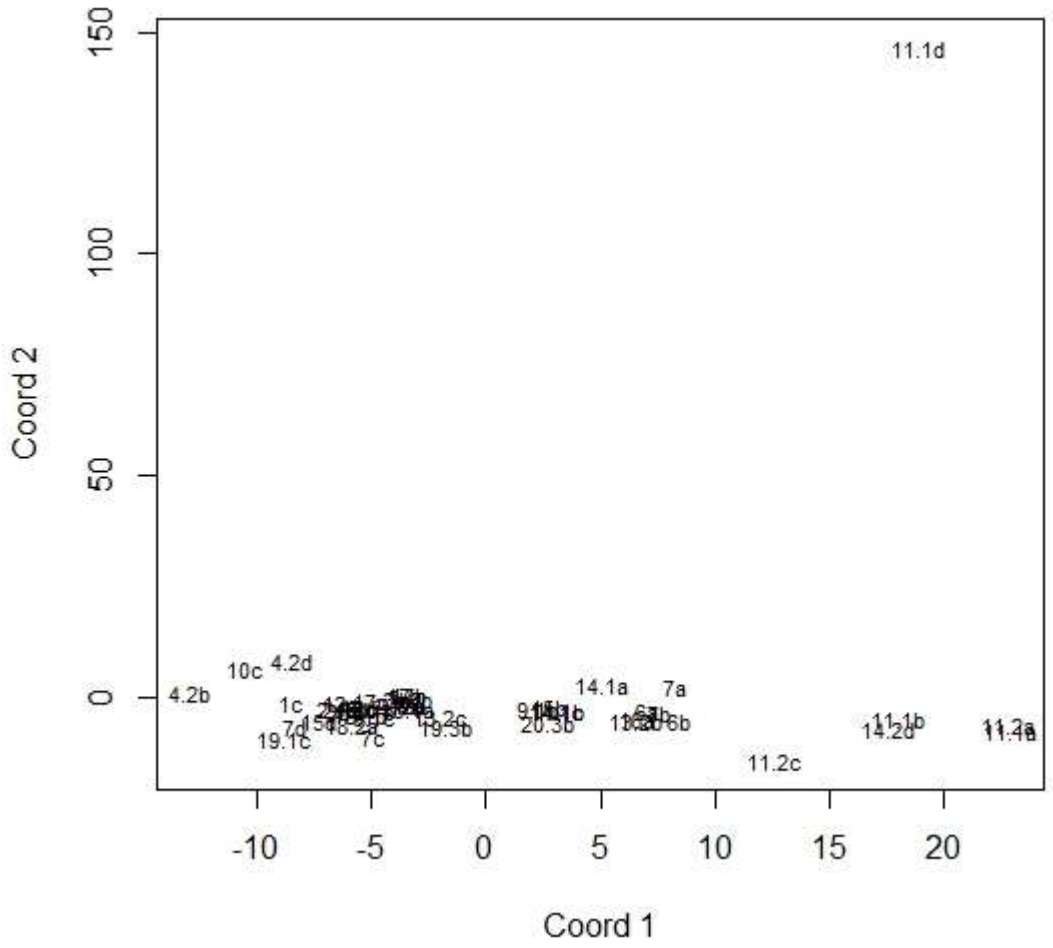
```
> plot(m[,1], m[,2], xlab="Coord 1", ylab="Coord 2", main="", type="n")
> text(m[,1], m[,2], labels = l, cex=.7)
```



Interpretation of a PCoA plot is straightforward. Objects close to each other are more similar on the basis of the distance matrix used as input. Plot 11.d is very different from the others; it is dominated by a certain grass species (Crawley 2012).

PCoA can only fully represent Euclidean components of the matrix even if the matrix contains non-Euclidean distances. The solution to overcome this limitation is given by the non-metric multidimensional scaling (NMDS), or using data transformations. **Non-metric multidimensional scaling** (NMDS) is based on rank orders instead of the absolute distances. It can be used for quantitative, semi-quantitative, qualitative, or mixed variables. It is a robust technique, e.g. tolerates missing distances. We use `isoMDS` from the "MASS" package (see also `sammon` in "MASS" for another approach). An iterative algorithm is used which starts from the result of `cmdscale` by default with the desired number of dimensions (2 by default). The algorithm minimizes the so called stress criterion to find the coordinates of the best spatial representation (best fit). Stress in general represents the extent to which the rank order of the fitted distances disagrees with the rank order of the observed dissimilarities. In practice, less than 5% stress is considered to suggest good fit.

```
> library(MASS)
> m = isoMDS(dist(d))
> m
initial  value 22.230754
iter   5 value 20.627186
iter  10 value 13.071990
iter  15 value 10.890715
iter  20 value 9.715061
iter  25 value 8.945729
iter  30 value 8.314638
iter  35 value 7.698929
iter  40 value 6.748821
iter  45 value 5.705382
iter  50 value 4.940255
final  value 4.940255
stopped after 50 iterations

> plot(m$points[,1], m$points[,2], xlab="Coord 1", ylab="Coord 2", main="",
type="n")
> text(m$points[,1], m$points[,2], labels = 1, cex=.7)
> m$stress
[1] 4.940255
```



MDS is often used to reveal the correlation pattern of the data (e.g. given in the form of correlation matrix). In this case, correlation must be transformed first to distance (e.g. apply `1 - cor` if `cor` denotes the correlation coefficient).

Factor analysis

Compared to PCA, the variables themself are of little interest here. The aim generally is to construct usable numerical values for properties that are hard to measure directly. We try to explore the assumed common factors behind the correlations of variables. In contrast to PCA, each factor contains a contribution from each variable (loadings, see `?loadings`), so the length of the factor is the number of variables. Note that "factor" here is not the same as the categorical variable we are using through the book.

There are different approaches for factor analysis. We demonstrate the `factanal` from the stats package. It is necessary to specify the number of factors we are interested in. For example let us start with 8.

```
> factanal(d,8)
```

```
Call:
factanal(x = d, factors = 8)

Uniquenesses:
   AC    AE    AM    AO    AP    AS    BH    BM    CM    CN    DG    FR    HL
0.005 0.005 0.387 0.764 0.131 0.053 0.342 0.421 0.731 0.649 0.310 0.005 0.665
   HP    HS    KA    LC    LH    LM    LO    LP    PL    PP    PT    RA    RC
0.238 0.096 0.541 0.399 0.281 0.470 0.275 0.206 0.162 0.558 0.210 0.005 0.383
   SM    TF    TG    TO    TP
0.676 0.492 0.508 0.352 0.106

Loadings:
   Factor1 Factor2 Factor3 Factor4 Factor5 Factor6 Factor7 Factor8
AC -0.454  -0.326  -0.208   0.788
AE  0.944  -0.204          -0.183          -0.120
AM -0.179   0.368           0.546   0.287   0.162  -0.188
AO -0.313  -0.197  -0.122  -0.102  -0.119  -0.120          -0.205
AP  0.850  -0.261          -0.194          -0.148
AS  0.779  -0.115   0.138  -0.148   0.109  -0.111   0.510
BH  0.365                                          0.715
BM -0.156   0.723  -0.114          -0.109
CM -0.167           0.409   0.162                          -0.186
CN -0.333                   0.311  -0.243   0.257          -0.111
DG  0.767  -0.138          -0.109           0.194          -0.133
FR -0.366          -0.242   0.119          -0.183           0.866
HL         -0.161          -0.480                          -0.257
HP          0.856   0.107
HS  0.732  -0.238   0.143  -0.272   0.268           0.360
KA -0.132   0.615  -0.151           0.114  -0.143
LC -0.311   0.535  -0.240   0.279          -0.277
LH -0.226   0.738  -0.168                   0.280
LM                         0.117           0.709
LO -0.157   0.804  -0.142                  -0.118
LP         -0.171   0.852                  -0.145
PL          0.182   0.694           0.434   0.354
PP  0.342  -0.163   0.174  -0.387   0.295           0.151
PT  0.799  -0.172                   0.157           0.286
RA  0.133           0.334           0.916           0.154
RC  0.504  -0.348           0.207   0.196   0.106           0.369
SM          0.553
TF          0.522   0.200           0.382          -0.150   0.123
TG                  0.534                   0.436
TO  0.209  -0.200   0.662  -0.324   0.100
TP                  0.893           0.242  -0.169

                Factor1 Factor2 Factor3 Factor4 Factor5 Factor6 Factor7 Factor8
SS loadings       5.401   4.504   3.423   1.909   1.725   1.344   1.136   1.130
Proportion Var    0.174   0.145   0.110   0.062   0.056   0.043   0.037   0.036
Cumulative Var    0.174   0.320   0.430   0.492   0.547   0.591   0.627   0.664

Test of the hypothesis that 8 factors are sufficient.
The chi square statistic is 321.71 on 245 degrees of freedom.
The p-value is 0.000718
```

For example, on factor 1, strong positive correlations are found with AE and AP and negative correlations with AC and FR. It was interpreted as this factor captures the gradient from the neutral to the acidic grasslands. Factor 2 captured the associations with the soil pH; factor 3 picks out the key nitrogen-fixing species and so on (Crawley 2012).

Cluster analysis

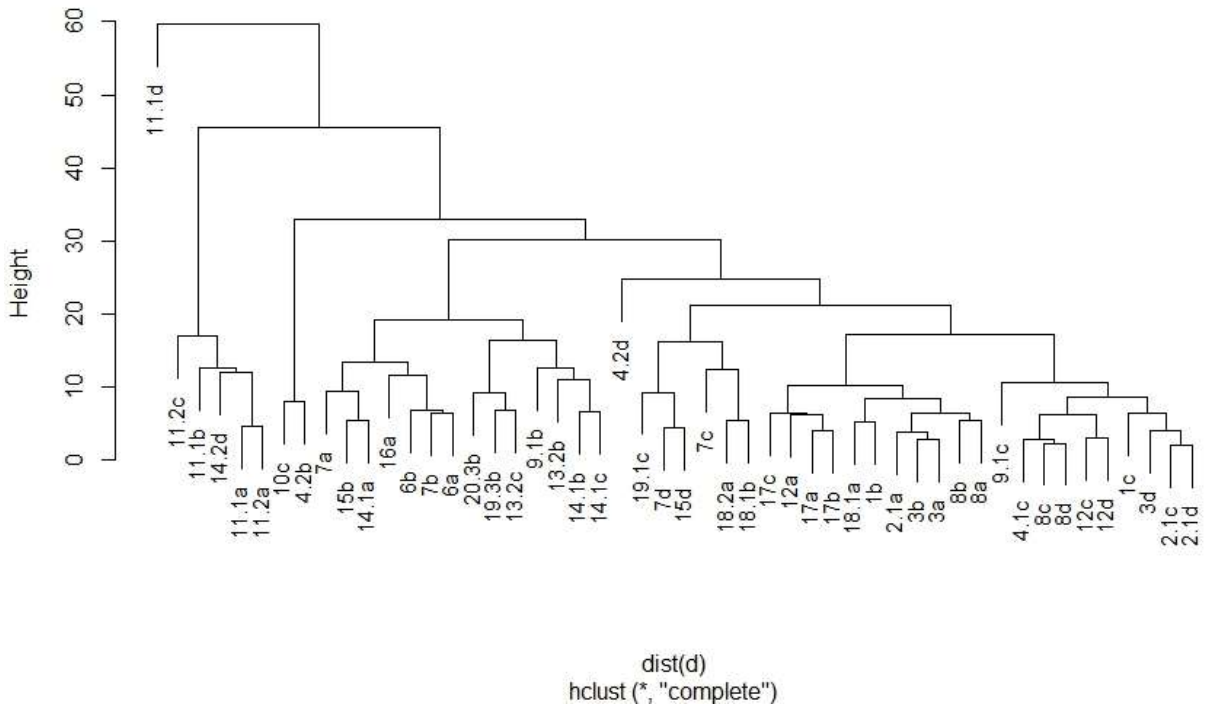
The aim of the cluster analysis is to separate groups (clusters) within the data on the basis of the similarity in variables, even if there is a redundancy within the latter ones. There are many different approaches that fall into one of the following main types:

- hierarchical classification first considers each object as a separate entity and ends up with a single cluster;
- divisive classification starts with a single aggregate and splits up clusters until all the objects represent different groups;
- partitioning methods try to allocate objects into predefined number of groups.

Hierarchical cluster analysis algorithms proceed iteratively. Initially each object is assigned to its own cluster. Then, it joins the most similar clusters step by step until a single cluster is obtained. Different methods exist for the agglomeration, i.e. the algorithm for calculation of distances between clusters. We will use the function `hclust` (from stats package), where the default method is the "complete linkage" (see `?hclust`). Another widely used approach is the UPGMA (Unweighted Pair Group Method with Arithmetic Mean, `hclust(method="average")`). Another key question is defining the similarity of two objects. Similarity is deduced from multivariate distance.

Which plots are the most similar in their botanical composition? We calculate the distance matrix, perform the cluster analysis and display the result as a dendrogram using the new labels. The plot can be saved in vector format (see e.g. `?pdf`) and it can be magnified in appropriate viewers (e.g. Acrobat reader).

```
> hc = hclust(dist(d))
> plot(hc,labels=l,main="")
```



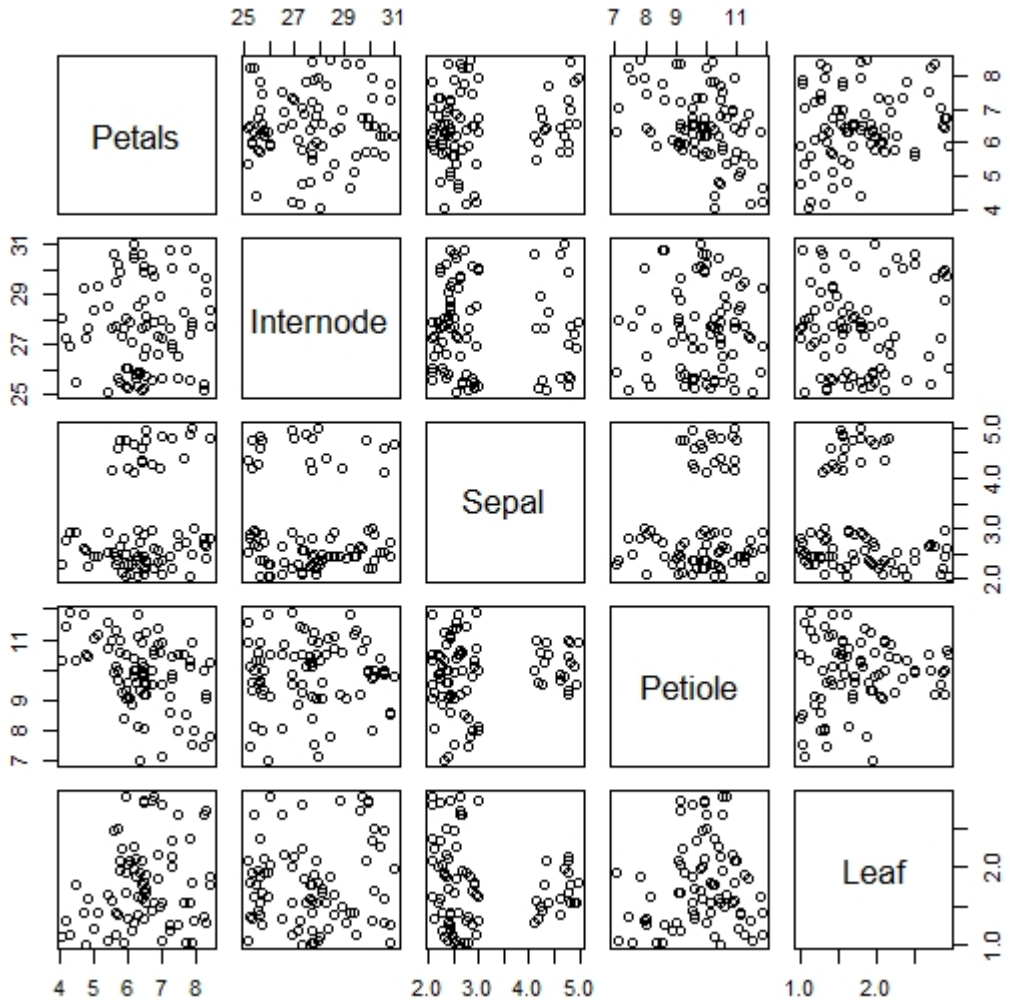
It was interpreted as (Crawley 2012) the highest breaks separate plots dominated by a given grass species (e.g. 11.1d) and high nitrogen plots also receiving phosphorus (e.g. plots 11). And so on.

Classification of taxonomic data provides a natural use of cluster analysis. We can test it on the "taxa" dataset.

```
> tx = read.table("planttax.txt",header=T)
> dim(tx)
[1] 80   5
> names(tx)
[1] "Petals"    "Internode" "Sepal"     "Petiole"   "Leaf"
```

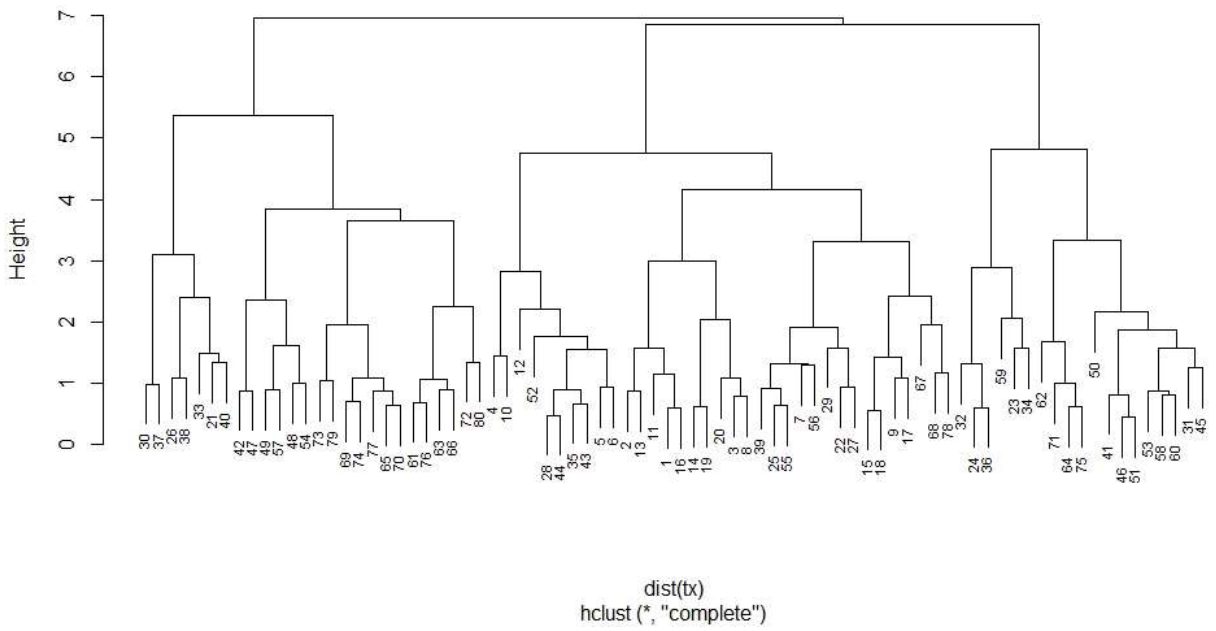
As a first look, we can plot every variable against each other.

```
> pairs(tx)
```



It seems that variables sepal length, petiole length and leaf width result in some subdivision. Let us repeat the hierarchical clustering on this data with the default distance and agglomeration method.

```
> plot(hclust(dist(tx)), main="")
```



We know the real groups in this example. For example we can realise that the rightmost split contains members from 3 different groups. The result seems to be unacceptable. Let us try other logics of classification.

Partitioning is another approach for clustering. One basic technique is implemented in `kmeans` from the stats package. The number of clusters fitted to the data must be given. In `kmeans` each object is simply assigned to the nearest centroid (the multidimensional equivalent of the mean of a group). Generally this number is unknown, but we know it for the taxa dataset.

```
> kmeans(tx,4)
K-means clustering with 4 clusters of sizes 22, 9, 16, 33

Cluster means:
    Petals Internode    Sepal   Petiole     Leaf
1 5.555707  27.97927 2.626847 10.518480 1.503062
2 7.803716  29.12466 2.593142  8.325013 1.416201
3 6.561599  29.80285 3.404990 10.128217 2.157938
4 6.651925  25.99485 3.195329  9.721562 1.846728

Clustering vector:
 [1] 1 1 1 4 4 4 1 1 1 4 1 4 1 1 1 1 1 1 1 1 4 4 2 2 1 4 1 4 2
[30] 2 3 2 4 2 4 2 2 4 1 4 3 4 4 4 3 3 4 4 4 3 3 4 3 4 1 1 4 3
[59] 2 3 4 3 3 3 4 3 3 1 4 4 3 4 4 4 3 4 4 1 4 4

Within cluster sum of squares by cluster:
[1]  53.81026  22.15085  43.90151 125.17553
 (between_SS / total_SS =  52.3 %)
```

Neither `kmeans` nor `hclust` is up to the job in this data. In general, when we do not know the group identities, then the result is often equivocal as illustrated by this artificial example.

Tree models

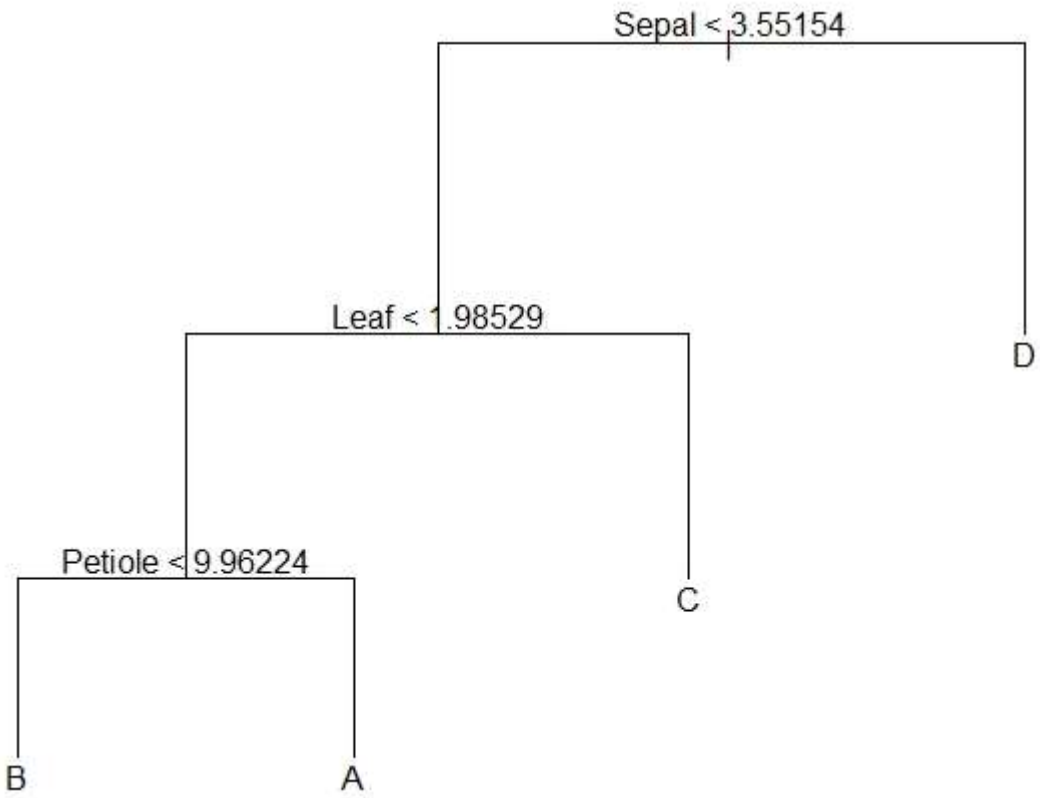
If we know the group identities of the objects, tree models can be efficient at creating keys based on the variables to allocate objects into the relevant categories. Tree models are computationally intensive methods that are used widely as a guidance to select important explanatory variables. Classification tree will be presented here from the "tree" package.

First we extend the "tx" dataframe with a "Taxon" column in order to include taxon names (denoted by A-D). "Taxon" is a four level categorical variable and its levels are the taxa. This is the response variable and we want to use five measurements (the explanatory variables) to separate the taxa.

```
> txy = cbind(Taxon=rep(c("A","B","C","D"),each=20),tx)
> names(txy)
 [1] "Taxon"     "Petals"    "Internode" "Sepal"     "Petiole"   "Leaf"
```

Now we can produce and plot the tree:

```
> library("tree")
> m = tree(Taxon~.,txy)
> plot(m)
> text(m)
```



Our aim was to find characters that explain most of the variation and finally we obtained a binary key for the separation of the taxa. Three of the five variables were selected. The key (from Crawley 2012):

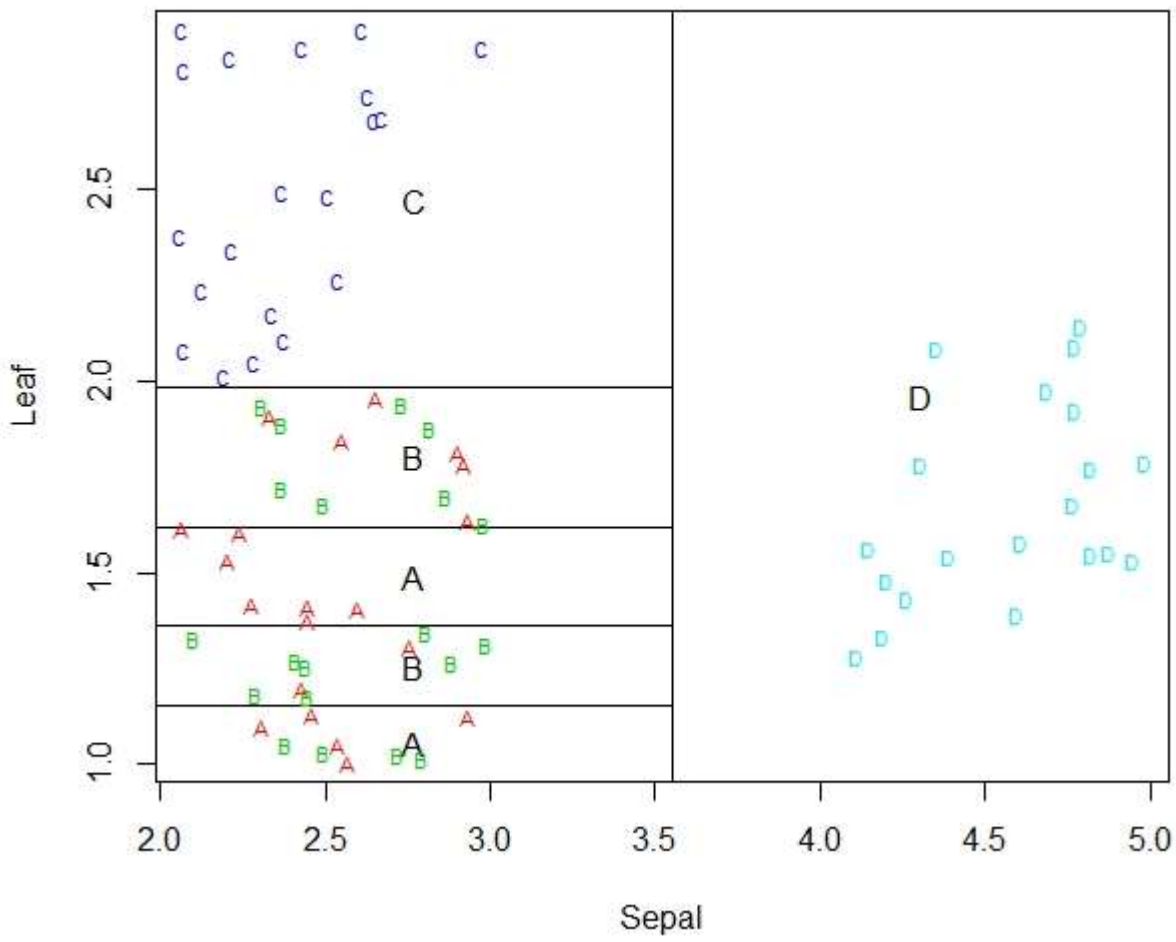
1. Sepal length >4.0	*Taxon D*
1. Sepal length ≤ 4.0	2.
2. Leaf width >2.0	*Taxon C*
2. Leaf width ≤ 2.0	3.
3. Petiole length < 10	*Taxon B*
3. Petiole length ≥ 10	*Taxon A*

This classification tree does much better than the multivariate techniques mentioned before. There is no error in the classification of the 80 sample individuals using the obtained key.

```
> summary(m)
Classification tree:
tree(formula = Taxon ~ ., data = txy)
Variables actually used in tree construction:
[1] "Sepal"   "Leaf"    "Petiole"
Number of terminal nodes:  4
Residual mean deviance:  0 = 0 / 76
Misclassification error rate: 0 = 0 / 80
```

Finally there is a plotting function for classification trees, for two explanatory variables. Selecting the two most important ones and repeating the analysis:

```
> m2 = tree(Taxon~Sepal+Leaf, txy)
> partition.tree(m2)
> with(txy, text(Sepal,Leaf,Taxon, cex=0.7, col=as.numeric(Taxon)+1))
```



It displays how the phase space has been divided up between the taxa. The last line added data points to the figure. Taxa C and D are separated by these two variables (A and B were separated by "Petiole" - but it is not used here).

Exercises

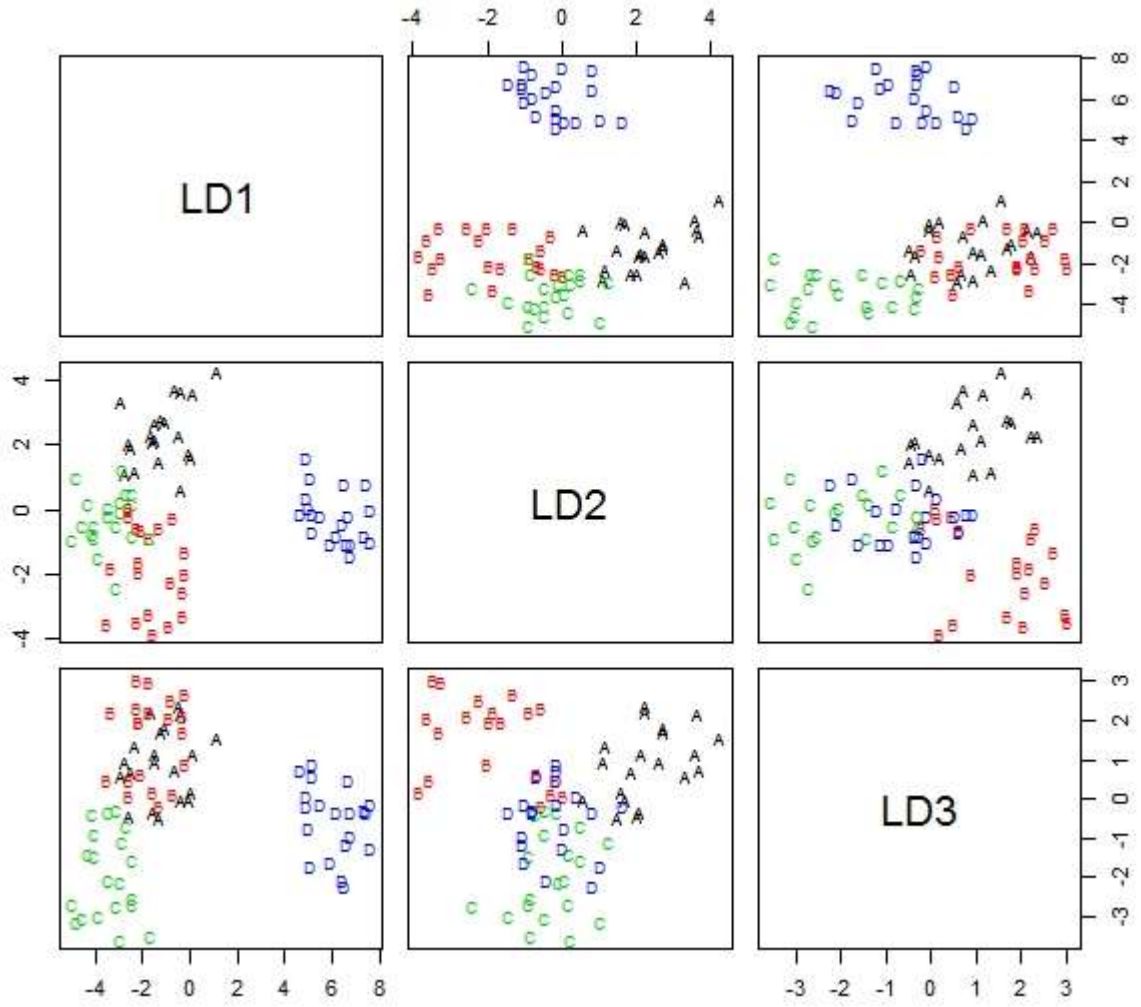
4. Repeat the hierarchical cluster analysis of the "plantdw" dataset using UPGMA with Euclidian distances and plot the dendrogram!
5. Display the group name on the dendrogram of the "taxa" dataset!
6. Does the result for "taxa" change with UPGMA?

Discriminant analysis

The aim of the discriminant analysis is to understand how the explanatory variables contribute to the correct classification. Grouping is known as in the case of tree models. For k groups we need k-1 discriminators, which are linear combinations of explanatory variables in linear discriminant analysis (LDA). The resulted model can be used to classify new data. LDA can be found in the "MASS" package.

```
> library("MASS")
```

```
> m = lda(Taxon~.,txy)
> plot(m, col=as.numeric(txy$Taxon))
```



Taxon D is separated completely by the linear discriminator LD1, while LD2 and LD3 are acceptable for Taxon B, LD1 and LD3 are good for taxon A. There is no clear separation for taxon C (Crawley 2012). The model:

```
> m
Call:
lda(Taxon ~ ., data = txy)

Prior probabilities of groups:
   A    B    C    D
0.25 0.25 0.25 0.25

Group means:
    Petals Internode    Sepal   Petiole     Leaf
A 5.424393  27.42605 2.528293 10.958002 1.460611
B 7.018869  27.68353 2.585029  8.452770 1.430544
C 6.724736  28.02338 2.369592  9.889299 2.498900
D 6.679908  27.48412 4.569818 10.159663 1.674055

Coefficients of linear discriminants:
                  LD1         LD2         LD3
```

```
Petals    -0.09252927 -0.45472160  0.02131069
Internode -0.05889892 -0.09634278 -0.05015767
Sepal      3.81477756 -0.37669969 -0.48275447
Petiole    0.21966707  1.37913523 -0.21228365
Leaf      -1.25251754 -0.56449276 -3.05274100

Proportion of trace:
   LD1    LD2    LD3
0.7627 0.1319 0.1054
```

As in the case of tree models, the key variables are "Sepal" (3.81, LD1), "Leaf" (-3.05, LD3) and "Petiole" (1.38, LD2). The prediction for classification using the discriminators:

```
> predict(m)
$`class`
 [1] A A A A A A A A A A A A A A A A A A A A B B B B B B B C B
[30] B B B B B B B B B B C C C C C C C C C C C C C C C C C C C
[59] C C D D D D D D D D D D D D D D D D D D D D
Levels: A B C D
...
```

One member of taxon B is misclassified (case 28).

Exercises

7. Create a random subset (e.g. for half of the samples) and repeat the LDA!
8. Classify the unused samples using the model obtained from the previous analysis!

Solutions

1. `plot(m,npcs=14)`
2. For example AC has positive and TF, PL have negative loadings.
3. `plot(d.full$pH, predict(m)[,2], xlab="soil pH", ylab="PC2")`. The second principal component is associated with declining soil pH.
4. `plot(hclust(dist(d),method="average"),labels=l)`
5. For example `l = rep(1:4,each=20); plot(hclust(dist(tx)),labels=l)`
6. Yes, but still it is bad, too (`plot(hclust(dist(tx),method="average"))`).
7. `train = sort(sample(1:80,40)); table(txy$Taxon[train]);`
   `m2 = lda(Taxon~., txy, subset=train); predict(m2)`
8. `predict(m2,txy[-train,])`

References

Crawley M. J. (2012) The R Book (2nd ed.). Wiley

Everitt B., Hothorn T. (2011) Introduction to Applied Multivariate Analysis with R. Springer

SUMMARY

Multivariate statisctics are used tho explore structure in the data.

- Reduce dimensionality with principal component, factor analysis or multidimensional scaling.
- Create groups of data with different types of cluster analysis or use recursive partitioning.
- Explore contribution of variables to a predefined classification using discriminant analysis.

R functions used in Chapter 10

biplot
cbind
cmdscale
dim
dist
factanal
hclust
isoMDS
kmeans
lda
library
names
pairs
partition.tree
paste
prcomp
rep
sort
str
table
text
tree
with

Chapter 11

Simulations

Simulations are used for many different purposes, for example investigating patterns in time and space. This chapter introduces a few techniques that demonstrate the power of R in this type of analysis. The first aim is to show how to generate pattern, with or without data in hand. It can be used to sharpen intuition and test the estimation toolbox. Our second aim is to show how to estimate statistical power using simulations. Selected examples of Bolker (2008) and Crawley (2012) will be reproduced.

Temporal population dynamics, the logistic map

Increase of the biological populations is typically a density-dependent process. The simplest model for this dynamics is known as the logistic map, defined here by the following quadratic recurrence equation:

$$N(t+1) = rN(t)(1-N(t)),$$

where N(t) is the population size at time t and the parameter r is the per-capita multiplication rate. We simulate and plot time series of populations for different values of r. First we write a function, called qm, to calculate the change of the population size from one generation to the next according to the above equation. The function has two arguments, the per-capita multiplication rate and actual scaled population size.

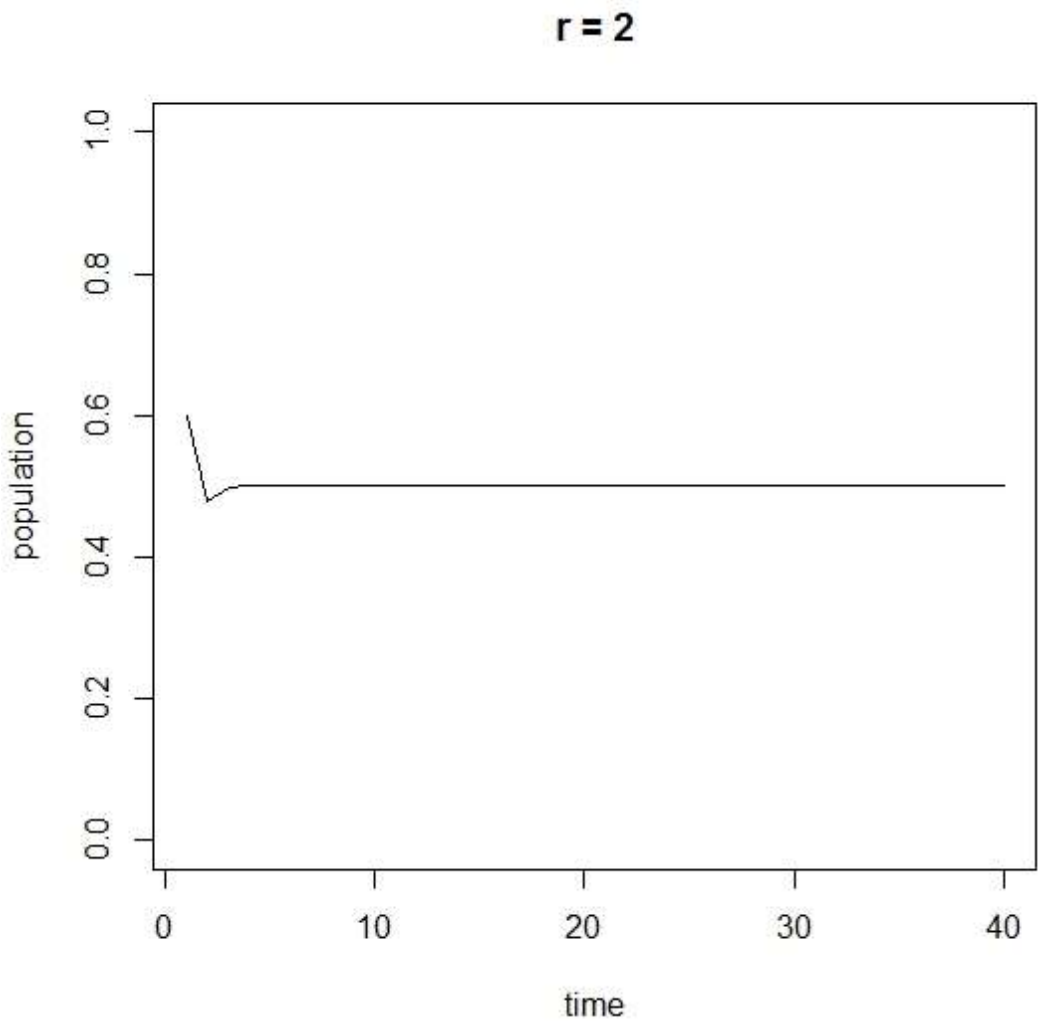
```
> qm = function(r,xt) r*xt*(1-xt);
```

We can use it similarly to the built-in R functions. For example:

```
> qm(1.3,0.4)
[1] 0.312
```

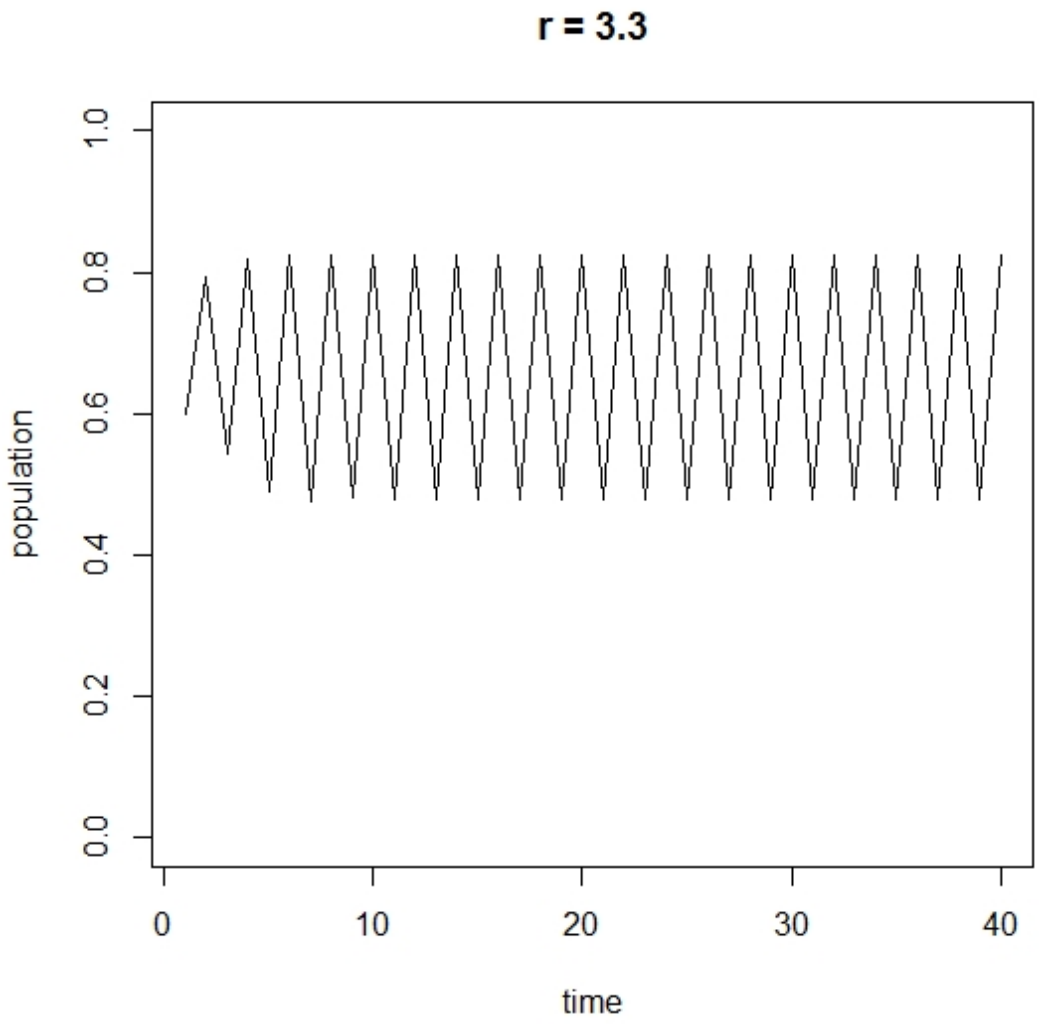
One way to go through the elements of a sequence is the use of `for` cycles. The index, denoted by "t" from time in the following example, represents the current element. Note that `t` is a function name and reserved in R, it stands for transpose basic function (see `?t`), but it is overwritten here. We will run four different simulations with different parameter values (r = 2; 3.3; 3.5; and 4).

```
> x1 = 0.6; maxt = 40; r = 2
> x = numeric(maxt)
> x[1] = x1; for(t in 2:maxt) x[t] = qm(r,x[t-1])
> plot(1:maxt,x,type="l",ylim=c(0,1),xlab="time",ylab="population",
+ main=paste("r = ",r,sep=""))
```



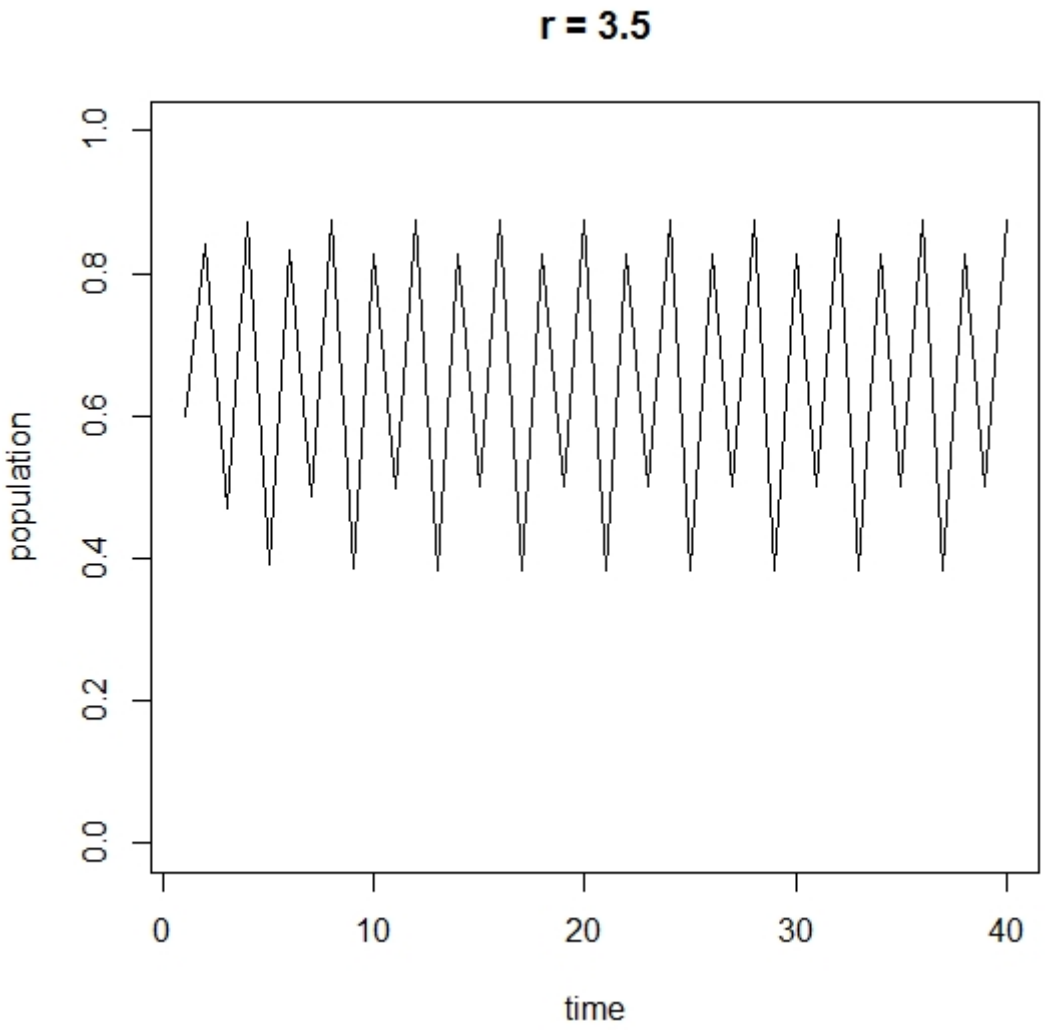
With r=2 the equilibrium (N=0.5) is reached quickly. This system has a stable equilibrium point.

```
> r = 3.3
> for(t in 2:maxt) x[t] = qm(r,x[t-1])
> plot(1:maxt,x,type="l",ylim=c(0,1),xlab="time",ylab="population",
+ main=paste("r = ",r,sep=""))
```



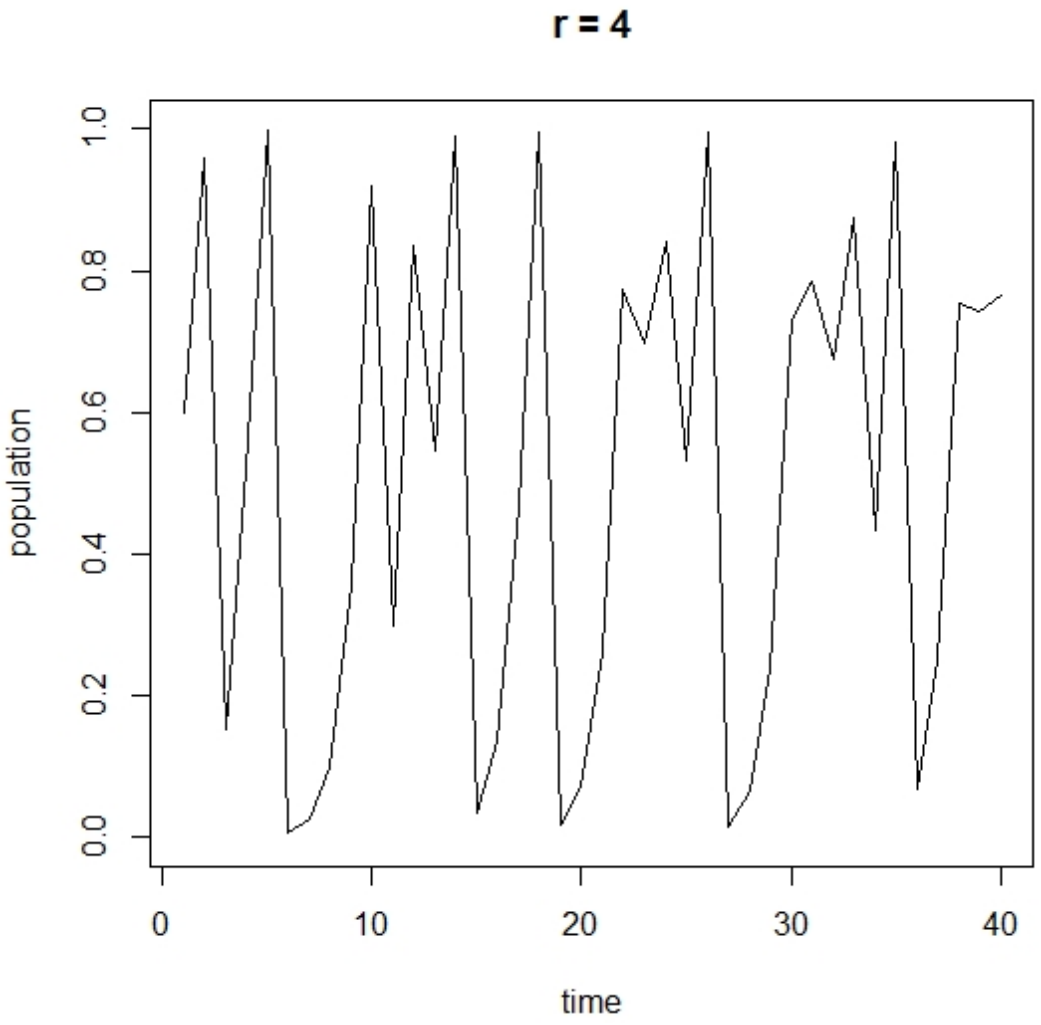
Now we have persistent two-point cycles.

```
> r = 3.5
> for(t in 2:maxt) x[t] = qm(r,x[t-1])
> plot(1:maxt,x,type="l",ylim=c(0,1),xlab="time",ylab="population",
+ main=paste("r = ",r,sep=""))
```



We have persistent four-point cycles.

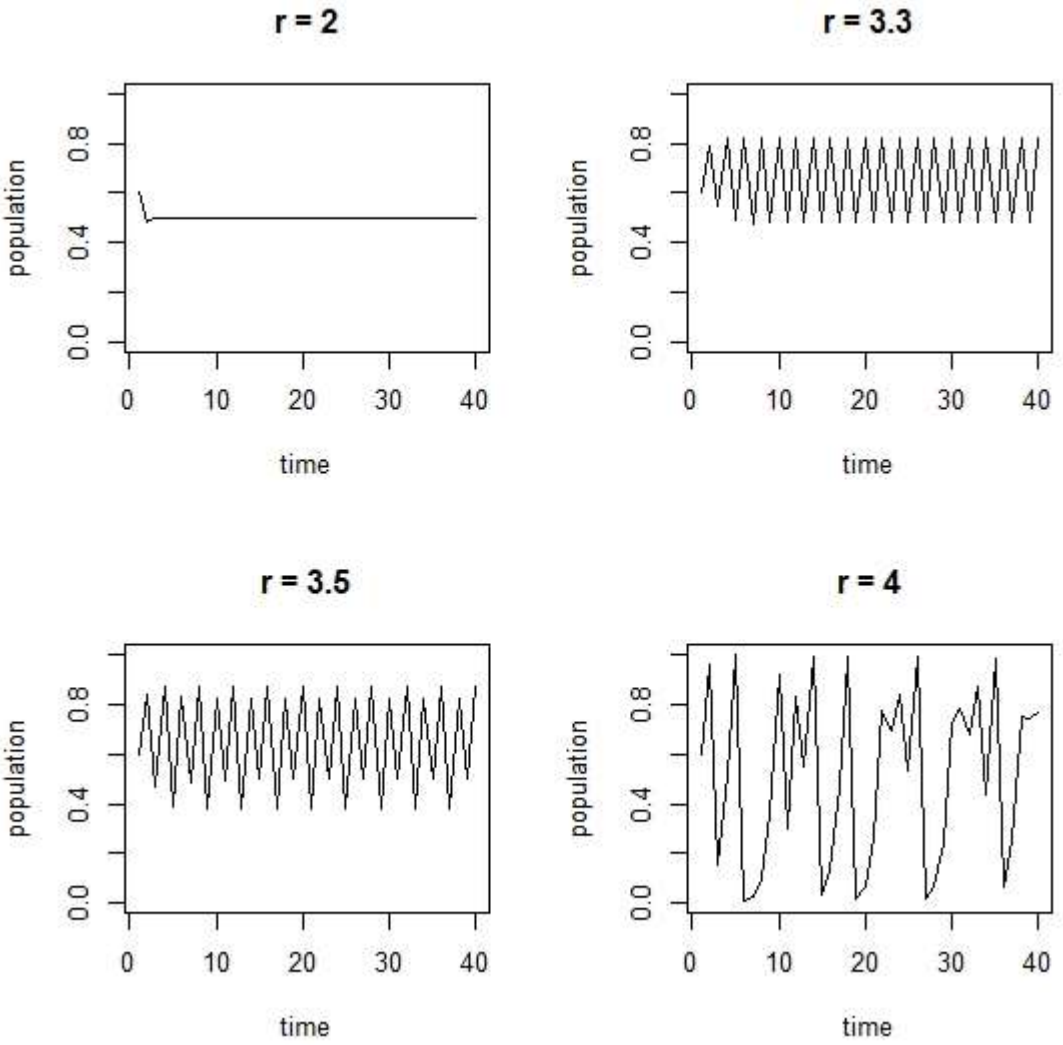
```
> r = 4.0
> for(t in 2:maxt) x[t] = qm(r,x[t-1])
> plot(1:maxt,x,type="l",ylim=c(0,1),xlab="time",ylab="population",
+ main=paste("r = ",r,sep=""))
```



The system has a chaotic behaviour.

Note that the results can be presented on one device, for example by using the `layout()` function before the first plot call. After the fourth plot command, we can reset the devise by `layout(matrix(1,1))`.

```
> layout(matrix(1:4,2,byrow=T))
> x1 = 0.6; maxt = 40; r = 2
> x = numeric(maxt)
> x[1] = x1; for(t in 2:maxt) x[t] = qm(r,x[t-1])
> plot(1:maxt,x,type="l",ylim=c(0,1),...
...
> plot(1:maxt,x,type="l",ylim=c(0,1),xlab="time",ylab="population",
+ main=paste("r = ",r,sep=""))
> layout(matrix(1:1,)
```

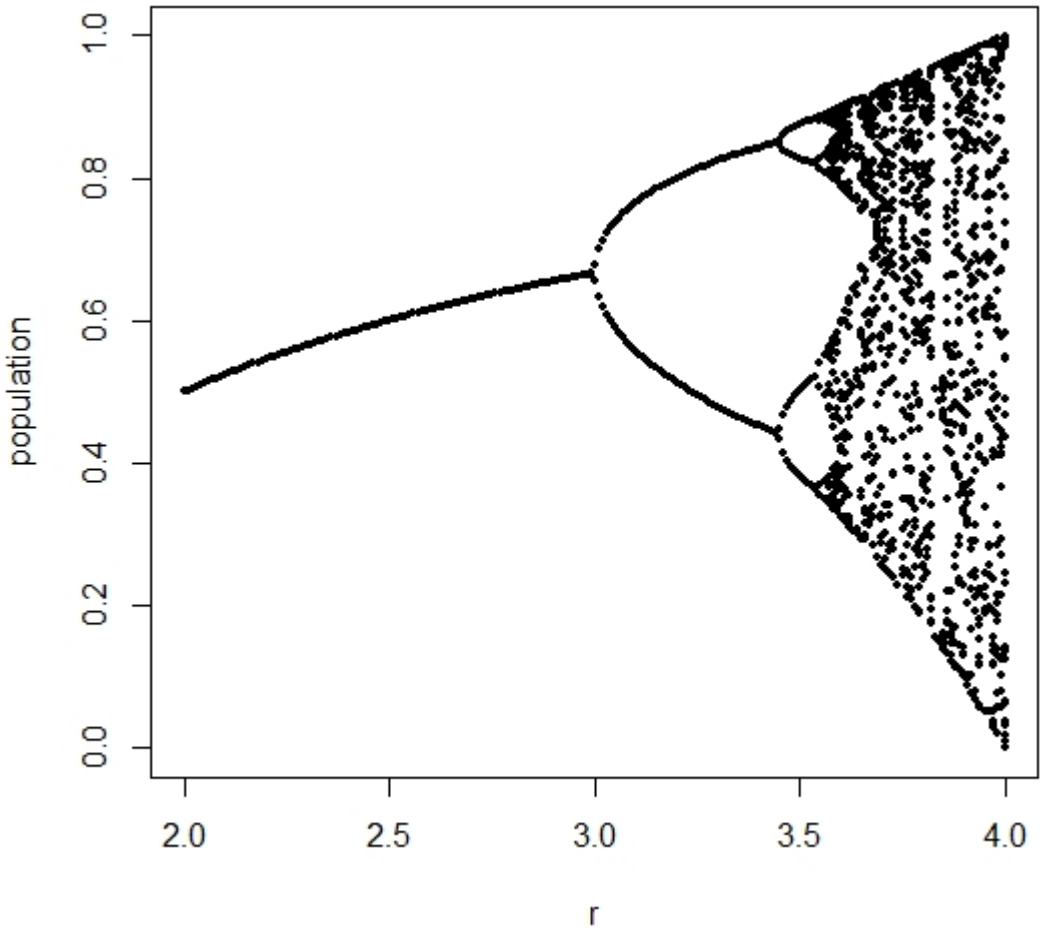


We created four snapshots for different values of r. But we can also investigate the behaviour of this system in more detail. Let us describe the dynamics as a function of the r parameter. We have to write a function that returns a set of sequential population densities (e.g. 30) following the transient period (e.g. 470 iterations).

```
> xts = function (r) {
+ x = numeric(500)
+ x[1] = 0.6
+ for (t in 2:500) x[t] = qm(r,x[t-1])
+ x[471:500]
+ }
```

We plot the 30 values against r within the interval we investigated above (from 2 to 4 with 0.01 step). A stable equilibrium point will be represented by one point (as all the 30 values are the same), two-point cycles by two point etc. This type of plot is called a bifurcation diagram.

```
> plot(c(2,4),c(0,1),type="n",xlab="r",ylab="population",main="")
> for(i in seq(2,4,0.01))
+ points(rep(i,30),sapply(i,xts),pch=16,cex=0.5)
```

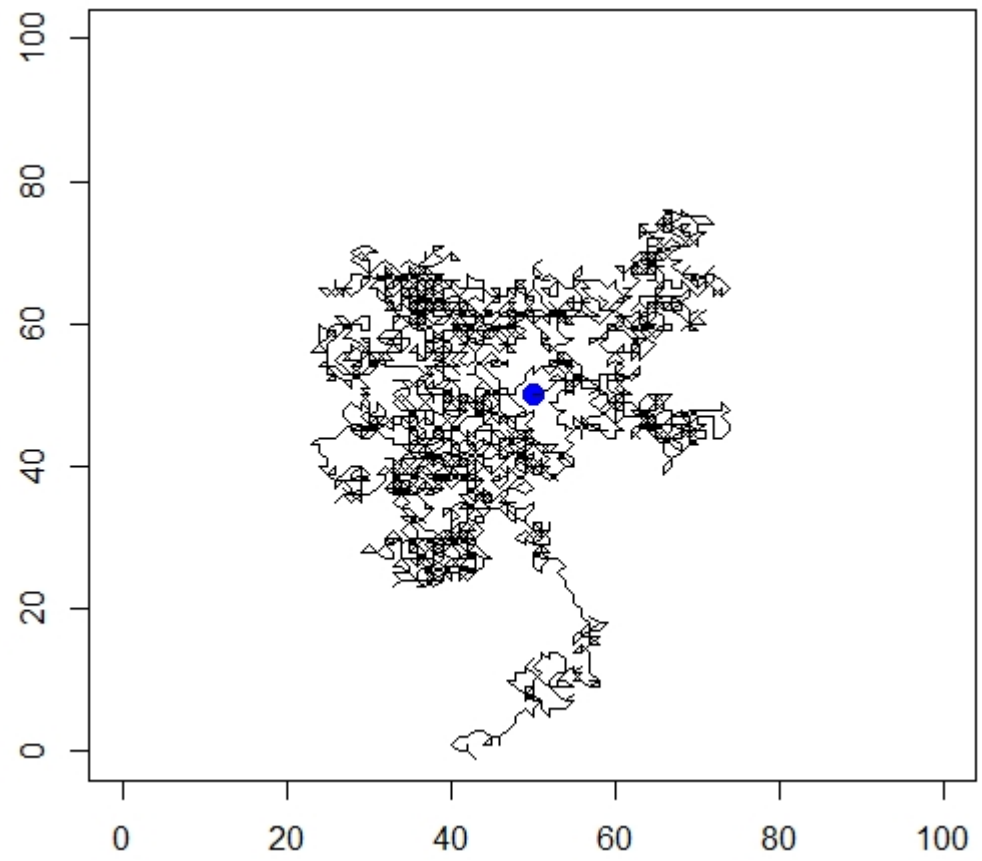


Pattern in space and time, the simulated random walk

We will follow an individual on a square surface (two dimensions) which scales from 0 to 100 units in both directions. The individual starts at the point (50,50) and leaves a trail. It can move one step in a time period in a given direction or stay (so the movement options are given by `c(-1,0,1)` in x and y directions, respectively). Let us suppose that all motions are equally likely.

In R, the `sample` function can be used to select one option randomly (with a specified probability, see `?sample`). We repeat this selection for x and y directions independently in each time period (say 10000). Note that margins should be handled in a more sophisticated spatially explicit simulation. It is stopped here when one of the margins is reached.

```
> plot(0:100,0:100,type="n",xlab="",ylab="")
> x = y = 50
> points(50,50,pch=16,col="blue",cex=1.5)
> for (i in 1:10000) {
+ xs = sample(c(1,0,-1),1)
+ ys = sample(c(1,0,-1),1)
+ lines(c(x,x+xs),c(y,y+ys))
+ x = x+xs; y = y+ys
+ if (x>100 | x<0 | y>100 | y<0) break
+ }
```



Stochastic simulation of the data

Previous examples presented one way of the use of simulations, where we had a model (based on a given theory) with fixed parameter values and worked forward to predict patterns qualitatively from a given initial condition. This approach is often called as forward modeling. In contrast, in inverse modeling we start from the data and work backward to choose a model and estimate parameters. When we have data, we can assess the similarity to a simulated pattern as a first step of the detailed analysis. Similarity may confirm that the model is reasonable. It also gives a rough estimate of the parameters. For static data, we can use functions to simulate the deterministic process (e.g. linear function) and add heterogeneity (e.g. from normal distribution).

Following Bolker (2008; more details are given in the book), the latter is illustrated on a simple example, where we have one group and one continuous covariate. We use a linear model with normally distributed errors. Data may represent productivity as a function of nitrogen concentration. More formally, let us specify Y random variable drawn from a normal distribution with mean a+bx and variance σ^2. It can be written as

$$y_i = a + bx_i + \varepsilon_i,\ \varepsilon_i \sim N(0,\sigma^2)\ .$$

The simulation starts by specifying the values of x (1, 2, ..., 20 in our case) and values for the parameters a (a=1) and b (b=2). We then calculate the deterministic part of the model (vector ydet).

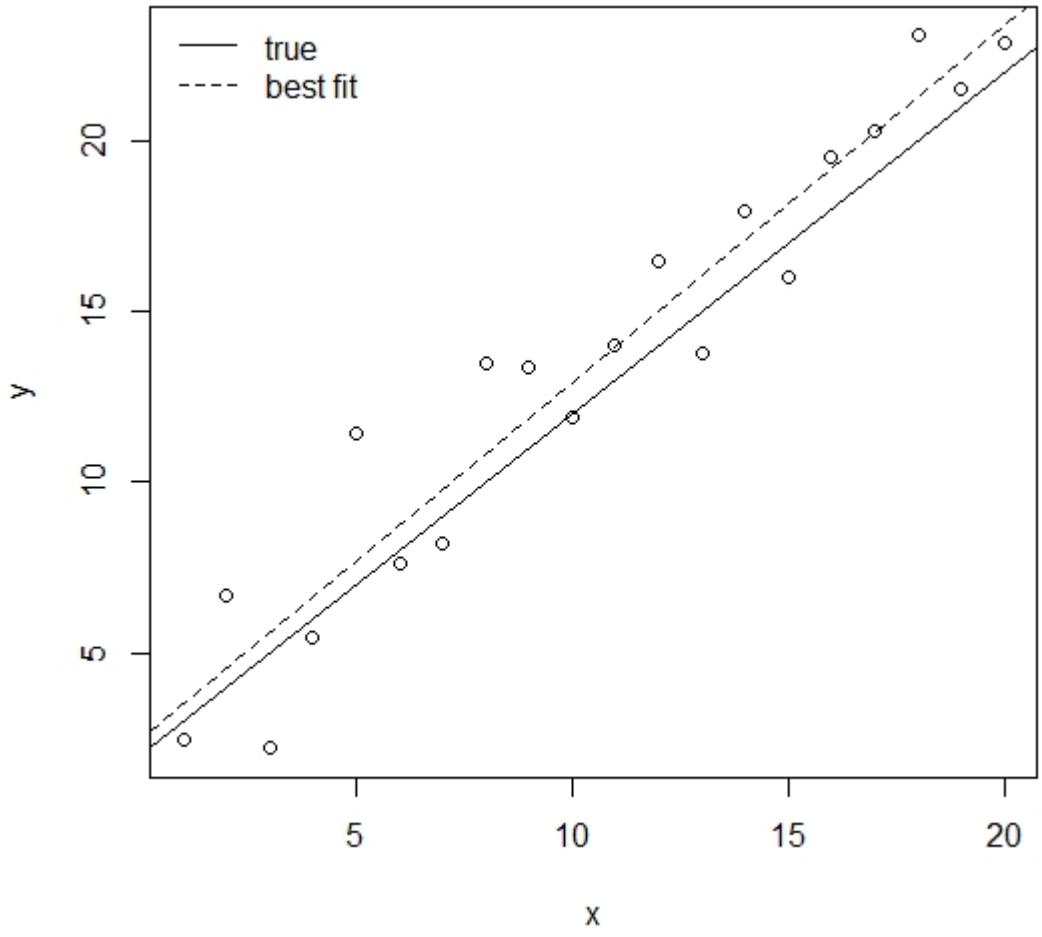
```
> x = 1:20
> a = 2; b = 1
> ydet = a+b*x
```

Next, we pick 20 random normal deviates (function `rnorm`). The mean is given by the deterministic part and standard deviation is fixed to 2, so the call `rnorm(20, mu=ydet, sd=2)` gives the desired result. Or, values of y can be obtained using the formula given above (the mean is zero by default in `rnorm`).

```
> y = ydet + rnorm(20, sd=2)
> plot(x,y)
```

Finally, we can show the theoretical relationship between x and y and the best-fit line by linear regression, `y =lm(y~x)`.

```
> abline(lm(y~x),lty=2)
> abline(a,b)
> legend("topleft",c("true","best fit"),lty=1:2,bty="n")
```



Randomness is incorporated in the simulation, so the theoretical ("true") and best-fit lines differ slightly.

This logic can be extended for many groups and covariates using more complex models. We can chain several functions and probability distributions easily in R. Note that the model usually cannot be expressed as a deterministic function plus error. Instead, we incorporate the

deterministic model as a control on one of the parameters of the error distribution (e.g. by specifying the `mu` argument as `mu=ydet)`.

Power analysis

Power analysis in a broad sense considers the following question: How do the quality and quantity of data and parameters affect the quality of the conclusions? Quality and quantity of the data and true properties of system include the

- number of observations,
- details of the experimental design (distribution of the data)
    - temporal and spatial extent and distribution (blocking, balance, etc.),
    - distribution of covariates,
- amount of variation,
- effect size.

For example large datasets are better, but there are trade-offs. It requires more effort, namely time and cost. Or, controlling extraneous variaton allows more powerful analysis - if we know what is "extraneous".

R has built-in functions for standard cases (see e.g. `power.t.test`) but more complicated, realistic situations require simulations. Simulations repeatedly generate random data based on the predefined model, then analyze each data set and count the proportion of results that are significant. That proportion is the estimated power for the model. There are packages that make the process simple (e.g. “paramtest” for general purpose).

For illustration we estimate the statistical power of detecting the linear trend in the data simulated in the previous section, as a function of the sample size. The elements of a single simulation step are the following:

- simulate the data points with a given sample size and model parameters
- test the hypothesis and store the result.

The R code for our example:

```
> x = 1:20
> a = 2; b = 1; sd = 8
> N = 20
> ydet = a+b*x
> y = rnorm(N,mean=ydet,sd=sd)
> m = lm(y~x)
> coef(summary(m))["x","Pr(>|t|)"]
[1] 0.006374994
```

We extracted the p-value of the desired parameter from the output matrix of `coef()`, that extracts the coefficients from the summary of the linear model fit, using matrix indexing by names.

```
> coef(summary(m))
            Estimate Std. Error  t value    Pr(>|t|)
(Intercept) 4.036208  3.9944745 1.010448 0.325671773
x           1.028948  0.3334523 3.085742 0.006374994
```

The p-value is less then the specified criterion (say $\alpha = 0.05$), so the null hypothesis is rejected. But we need the power, the probabily of succesfully rejecting the null hypothesis when it is false. To estimate the power we have to repeat the procedure many times (denoted as `nsim`) and simply calculate the proportion of runs when we reject the null hypothesis. The p-value of each simulation is stored in the `pval` vector. We set the number of simulations to 400 (a reasonable choice if we want to estimate percentage).

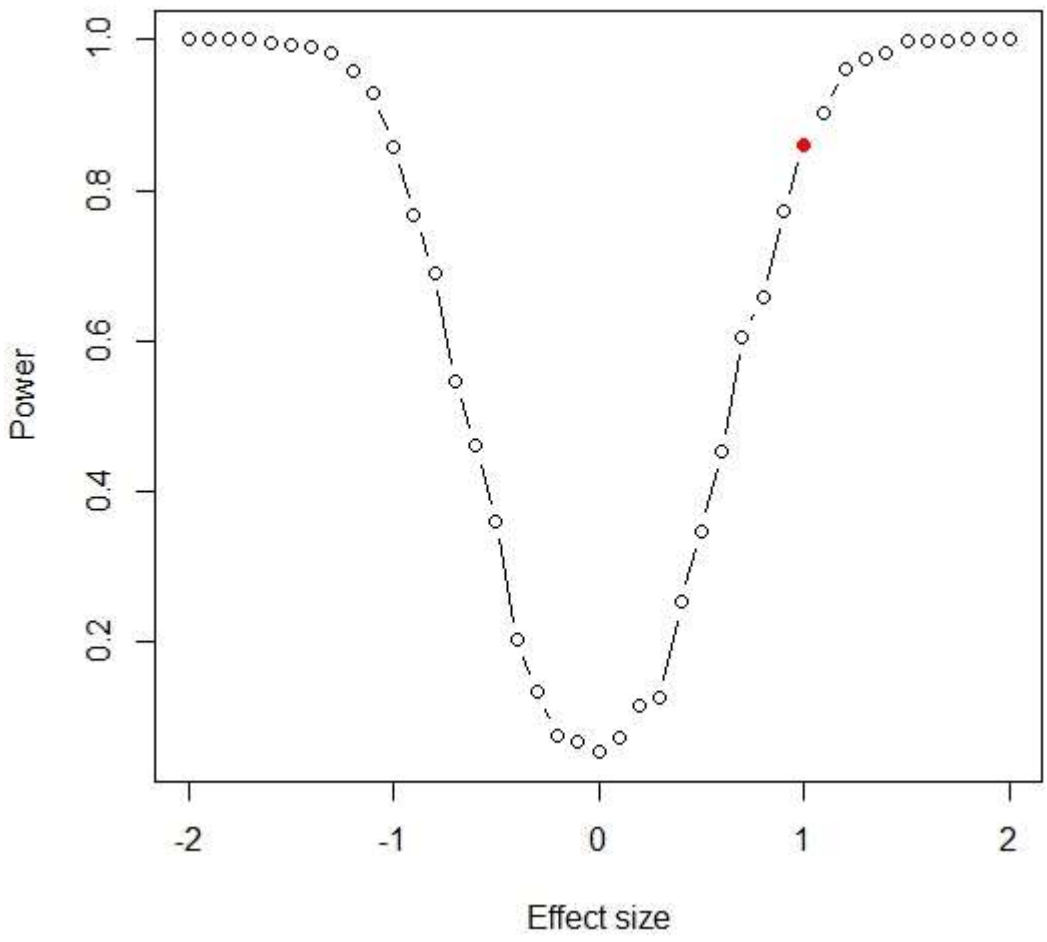
```
> nsim = 400
> pval = numeric(nsim)
> for (i in 1:nsim) {
+   ydet = a+b*x
+   y = rnorm(N,mean=ydet,sd=sd)
+   m = lm(y~x)
+   pval[i] = coef(summary(m))["x","Pr(>|t|)"]
+ }
> sum(pval<0.05)/nsim
[1] 0.86
```

Now we have the power of the single experimental design. But generally we want to know how the power changes as we modify the design. To do this, we have to repeat the last procedure many times, each time changing a given parameter such as the slope (or the distribution of x, etc.). For example for the slopes (stored in the vetor `power.b`):

```
> power.b = numeric(length(bvec))
> for (j in 1:length(bvec)) {
+   b = bvec[j]
+   for (i in 1:nsim) {
+     ydet = a+b*x
+     y = rnorm(N,mean=ydet,sd=sd)
+     m = lm(y~x)
+     pval[i] = coef(summary(m))["x","Pr(>|t|)"]
+   }
+   power.b[j] = sum(pval<0.05)/nsim
+ }
```

We can plot the power as a function of the effect size (the distance of the parameter from the null hypothesis value, actually it equals to the slope). Red dot shows the previous scenario (where b=1).

```
> plot(bvec,power.b,type="b",ylab="Power",xlab="Effect size")
> points(1,0.86,pch=16,col="red")
```



We could calculate the power for many parameter combinations (e.g. add sample size, standard deviation) using another `for` loops, and show the result using `contour`.

References

Bolker B. (2008) Ecological Models and Data in R. Princeton Univ Press

Crawley M. J. (2012) The R Book (2nd ed.). Wiley

SUMMARY

Simulation models are used for investigating patterns in time and space or for testing our estimation tools or power. Examples are presented in the R way.

R functions used in Chapter 11

abline
coef
function
layout
legend
for
lines
lm
numeric
points
rnorm
sample
sapply
seq
summary

Chapter 12

Phylogenetics

In traditional analyses species were treated as independent entities. In ecology, typical data matrices (data frames in R) contained the taxon name and abundances in different localities, perhaps together with explanatory variables. But independence is an invalid assumption. Species vary in their degree of relatedness and their functional similarity. Thus, analyses may require phylogenetic trees and ecological (or trait) matrices, too. The "Taxon" dataset used for multivariate analyses is an example of the latter (Chapter 10).

Phylogenetics deals with evolutionary relationships among different entities, including species and their properties. It provides the basic framework for biological reasoning. In statistical sense, the reconstruction of phylogeny is a special multivariate clustering method. This chapter introduces handling phylogenetic data in R: how it is structured and how it can be plotted and manipulated. We do not discuss the reconstruction workflow in detail. Paradis (2011) gives a nice introduction to phylogenies in R (together with comparative analyses). The package "ape" we use in this book is introduced also in Paradis (2011). Another aim of this chapter is to illustrate a way how DNA sequences can be handled in R.

We will reproduce some of the analyses from Chapters 3, 5, and 6 of Paradis (2011) for a gall wasp cytochrome c oxidase sequence dataset (a subset from Nylander et al. 2004). Many advantages of using R will be demonstrated. For example diagnostic methods can be carried out easily. The steps of the analysis:

1. Dowload the DNA sequences from the GenBank;
2. Sequence alignment;
3. Play with genetic distances, compare the models of sequence evolution;
4. Phylogenetic reconstruction, visualisation of the trees;

Obtaining the data

We first download sequences for 16 species from the GenBank database using the accession numbers (these are listed in Nylander et al. 2004).

```
> library(ape)
> x = c("AY368909", paste("AY3689", 10:15, sep = ""),
+ "AY368923", paste("AY3689", 29:31, sep = ""),
+ "AF395174","AF395176","AF395178","AF395181","AY368932")
> x
 [1] "AY368909" "AY368910" "AY368911" "AY368912" "AY368913"
 [6] "AY368914" "AY368915" "AY368923" "AY368929" "AY368930"
[11] "AY368931" "AF395174" "AF395176" "AF395178" "AF395181"
[16] "AY368932"

> cyn.seq = read.GenBank(x)
> cyn.seq
16 DNA sequences in binary format stored in a list.

Mean sequence length: 1077.75
```

```
   Shortest sequence: 1077
    Longest sequence: 1078

Labels:
AY368909
AY368910
AY368911
AY368912
AY368913
AY368914
...

Base composition:
    a     c     g     t
0.324 0.112 0.136 0.428
```

The length of the downloaded sequences differs.

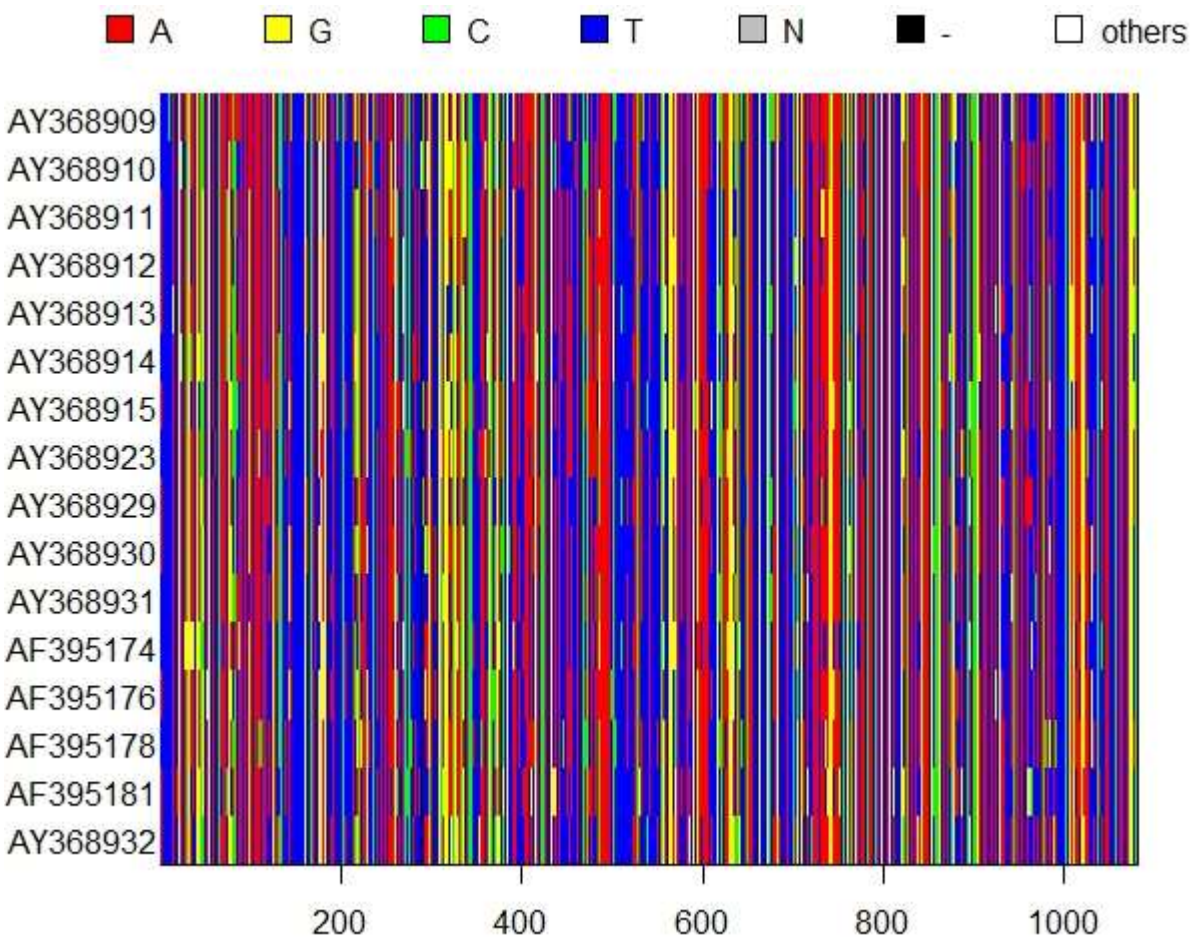
```
> table(sapply(cyn.seq, length))

1077 1078
   4   12
```

Four out of the 16 sequences are shorter by one nucleotide.

Sequence alignment

The next step is the alignment of the DNA sequences. We use Clustal here. Note that the external program must be in the PATH. Alternatively, you can use Clustal separately (later we will see how sequences can be saved in a standard format).

```
> cyn.ali = clustal(cyn.seq)

CLUSTAL 2.1 Multiple Sequence Alignments


Sequence format is Pearson
...
FASTA file created!

Fasta-Alignment file created ...

> image(cyn.ali)
```



Note that some other alignment programs can be called in a similar way from R (e.g. mafft for MAFFT in the phyloch package).

For further analysis we need the aligned sequences, the name of the taxa and accession numbers exclusively. We will collect the names into a vector associated with the accession numbers, check them and drop all unnecessary information.

```
> taxa.cyn = attr(cyn.seq, "species")
> names(taxa.cyn) = names(cyn.seq)
> rm(cyn.seq)
> taxa.cyn
                    AY368909                     AY368910
      "Synergus_crassicornis"            "Ceroptres_cerri"
                    AY368911                     AY368912
  "Synophromorpha_sylvestris"    "Xestophanes_potentillae"
                    AY368913                     AY368914
       "Diastrophus_turgidus"    "Diastrophus_potentillae"
                    AY368915                     AY368923
      "Liposthenes_glechomae"            "Aylax_papaveris"
                    AY368929                     AY368930
           "Pediaspis_aceris"      "Neuroterus_numismalis"
                    AY368931                     AF395174
           "Biorhiza_pallida"           "Diplolepis_rosae"
                    AF395176                     AF395178
           "Andricus_kollari" "Plagiotrochus_quercusilicis"
                    AF395181                     AY368932
       "Periclistus_brandtii"           "Parnips_nigripes"
```

Correction in names (e.g. remove the subspecies name) can be done easly in this stage if it is necessary.

We may also have additional biological data for these species. It is stored in the "cyndata.txt" file.

```
> cyn.data = read.table("cyndata.txt")
> str(cyn.data)
'data.frame':   16 obs. of  3 variables:
 $ Host   : Factor w/ 8 levels "Acer","Glechoma",..: 5 5 6 8 4 7 4 ...
 $ Biology: Factor w/ 3 levels "galler","inquiline",..: 2 2 2 2 1 1 1  ...

> head(cyn.data)
                            Host   Biology
Synergus_crassicornis       Quercus inquiline
Ceroptres_cerri             Quercus inquiline
Periclistus_brandtii           Rosa inquiline
Synophromorpha_sylvestris     Rubus inquiline
Xestophanes_potentillae   Potentilla    galler
Diastrophus_turgidus        Rosaceae    galler
```

The species names are given as rownames in the `cyn.data` dataframe. Compare the species names to the names obtained from the GenBank:

```
> x = match(as.character(taxa.cyn),rownames(cyn.data))
> x
 [1]  1  2  4  5  6 NA  8  9 10 11 12 13 14 15  3 16
> as.character(taxa.cyn)[is.na(x)]
 [1] "Diastrophus_potentillae"
```

The last function returned the name of the aligned sequence that cannot be matched. After reversing the matching we see that synonymous names were used.

```
> rownames(cyn.data)[is.na(match(rownames(cyn.data),
+ as.character(taxa.cyn)))]
[1] "Gonaspis_potentillae"
```

Let us replace the name of the downloaded sequence (it is in the sixth position of the `taxa.cyn` character vector, or you can use `taxa.cyn["AY368914"]`):

```
> taxa.cyn[6] = "Gonaspis_potentillae"
```

Finally, we save together the aligned sequences, species names and additional data objects for later work.

```
> save(cyn.ali, taxa.cyn, cyn.data, file = "cyn.RData")
```

It can be reloaded any time by the function call `load("cyn.RData")`.

We can save the aligned sequences to a text file in a standard format. By default an interleaved "phylip" formatted file is created (see also `?write.FASTA`).

```
> write.dna(cyn.ali,"cyn.txt")
```

Genetic distances

The next step is to compare genetic distances using different models of sequence evolution (see `?dist.dna` for the implemented models and other options). First, distance matrices are calculated from the aligned sequences. We use pairwise deletion option as we have incomplete sequences.

```
> cyn.K80 = dist.dna(cyn.ali, pairwise.deletion = TRUE)
> cyn.F84 = dist.dna(cyn.ali, model = "F84", p = TRUE)
> cyn.TN93 = dist.dna(cyn.ali, model = "TN93", p = TRUE)
> cyn.GG95 = dist.dna(cyn.ali, model = "GG95", p = TRUE)
```

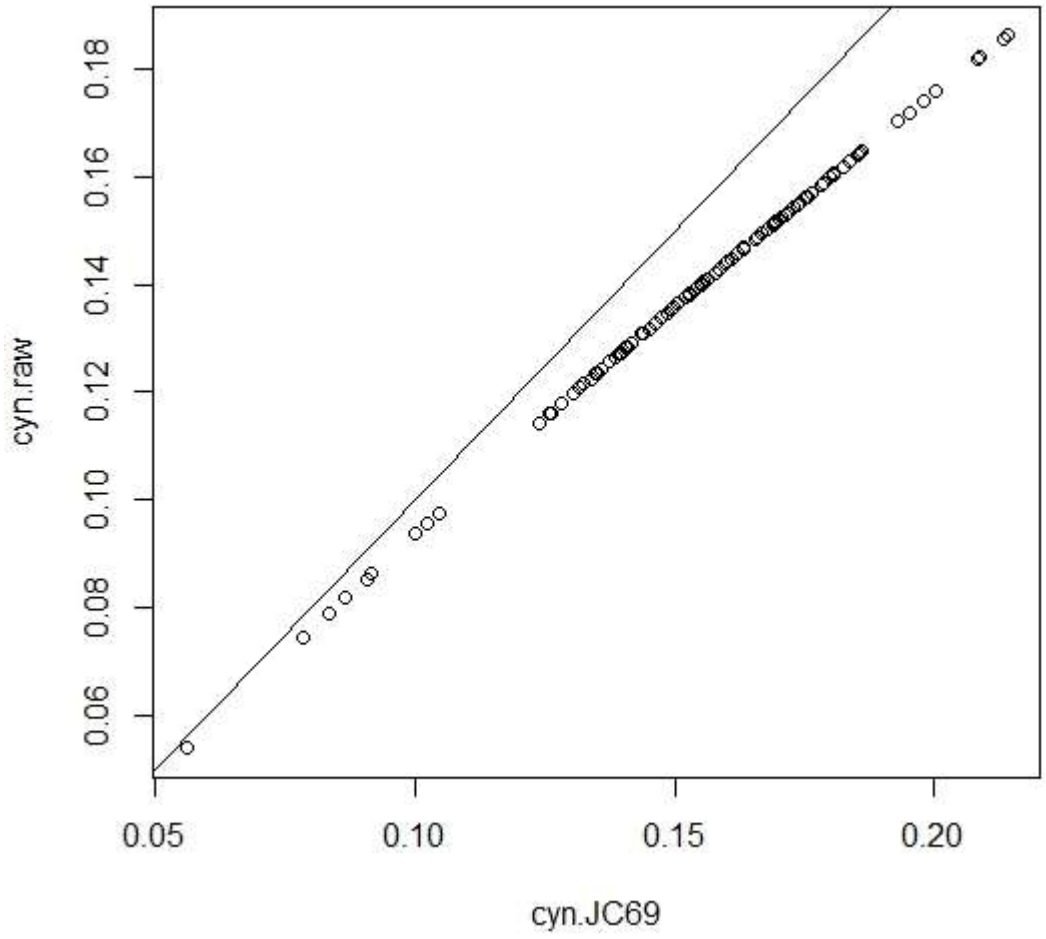
Correlation among the models gives some impression about their similarity.

```
> round(cor(cbind(cyn.K80, cyn.F84, cyn.TN93, cyn.GG95)), 3)
         cyn.K80 cyn.F84 cyn.TN93 cyn.GG95
cyn.K80    1.000    1.00    1.000    0.932
cyn.F84    1.000    1.00    1.000    0.930
cyn.TN93   1.000    1.00    1.000    0.929
cyn.GG95   0.932    0.93    0.929    1.000
```

Model GG95 differs considerably from the others.

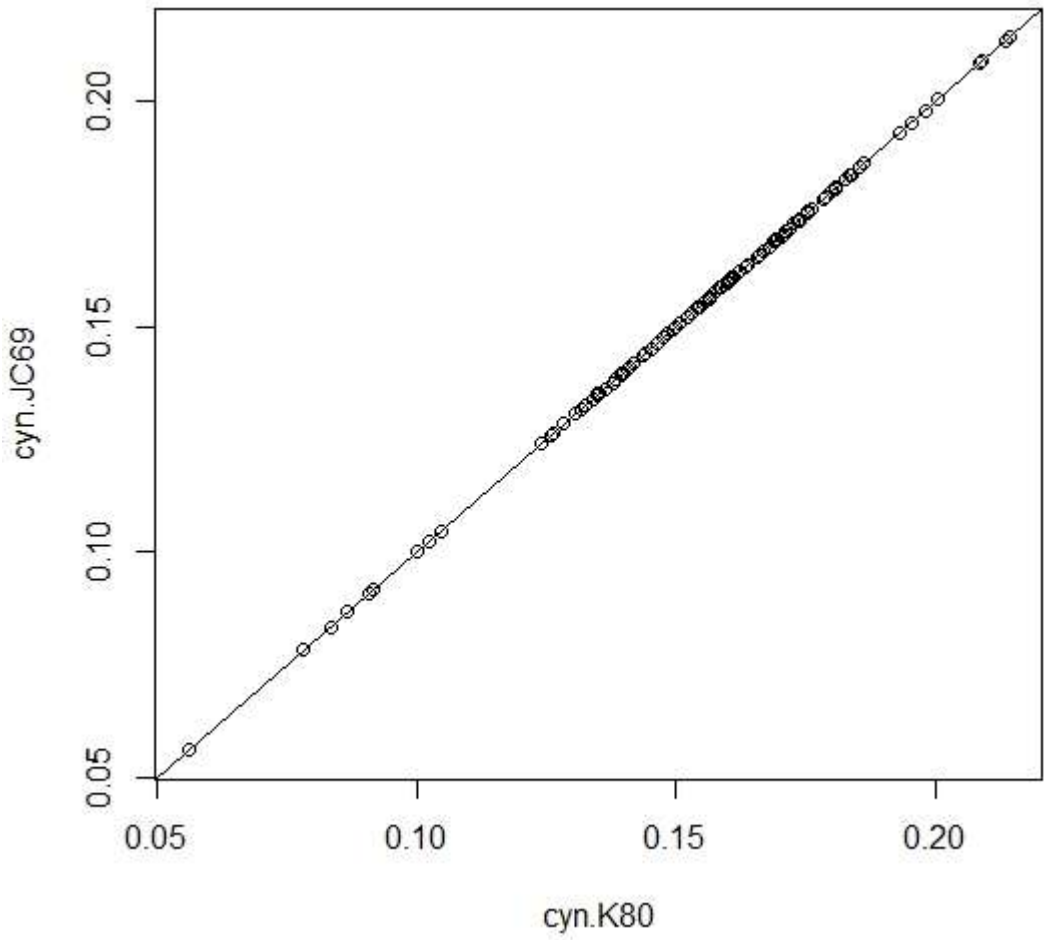
Saturation of substitutions can be investigated graphically. For example, comparing the observed pattern (the p-distance, proportion of different sites) to the simple Jukes-Cantor (JC69) model informs us about the influence of multiple substitutions.

```
> cyn.JC69 = dist.dna(cyn.ali, model = "JC69", p=TRUE)
> cyn.raw = dist.dna(cyn.ali, model = "raw", p=TRUE)
> plot(cyn.JC69, cyn.raw); abline(b = 1, a = 0)
```



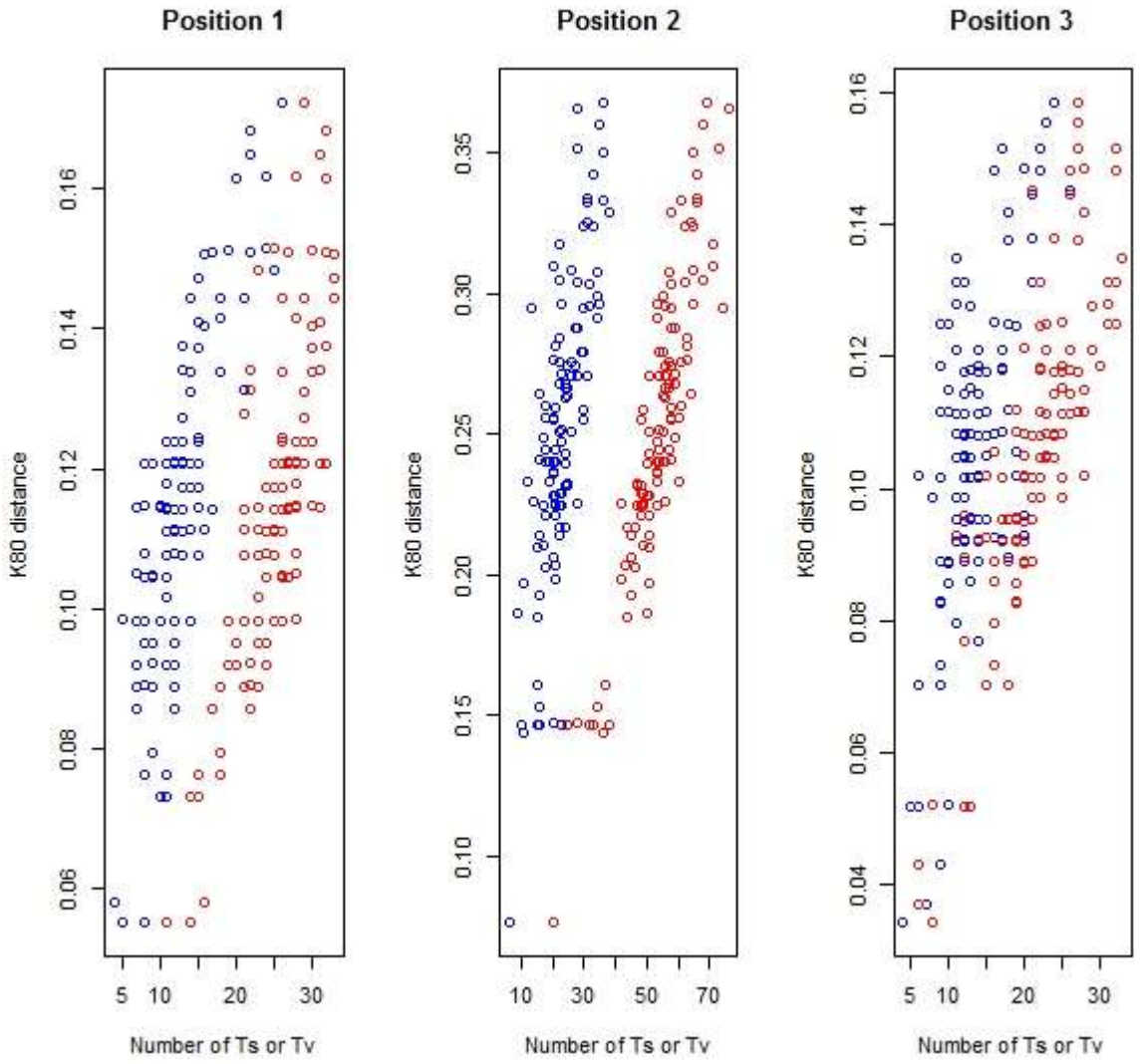
Some saturation is expected for the most divergent sequences. Comparing Kimura one parameter model (K80) to Jukes-Cantor suggests no effect of the transition/transversion ratio.

```
> plot(cyn.K80,cyn.JC69); abline(b = 1, a = 0)
```



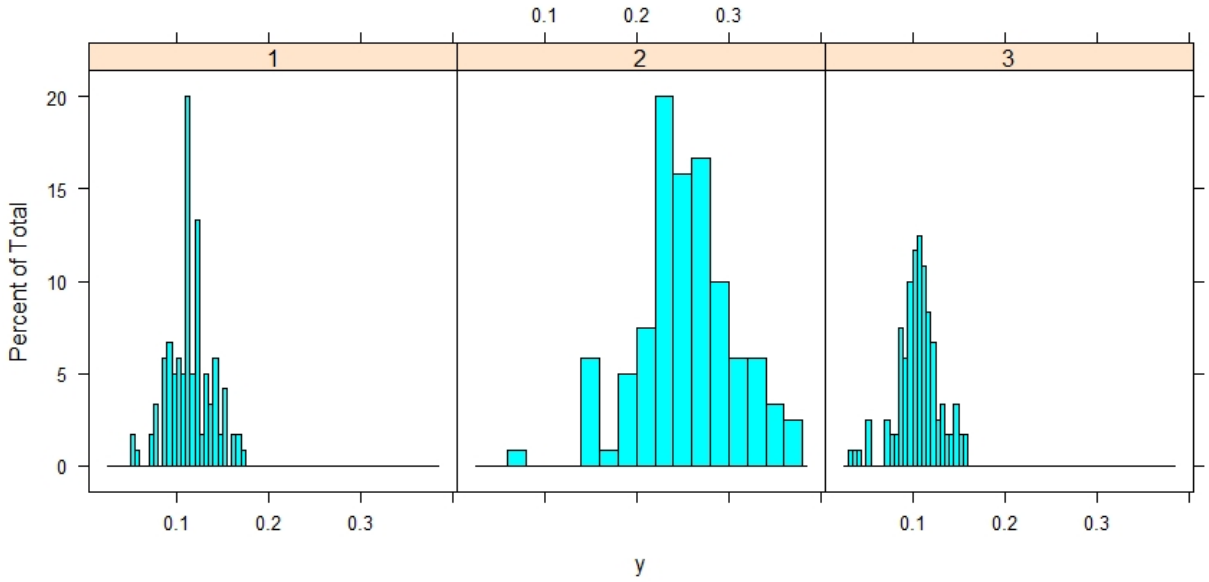
We can create the traditional "saturation plot" for each codon positions separately (number of transitions and transversions against the K80 distance). Note that postion 1 here is the starting position of the sequence.

```
> layout(matrix(1:3, 1))
> for (i in 1:3) {
+ s = logical(3); s[i] = TRUE
+ x = cyn.ali[, s]
+ d = dist.dna(x, p = TRUE)
+ ts = dist.dna(x, "Ts", p = TRUE)
+ tv = dist.dna(x, "Tv", p = TRUE)
+ plot(ts, d, xlab = "Number of Ts or Tv", col = "blue",
+ ylab = "K80 distance", xlim = range(c(ts, tv)),
+ main = paste("Position", i))
+ points(tv, d, col = "red")
+ }
> layout(matrix(1,1))
```



Histogram of the pairwise distances for each codon position shows further details, namely it suggests something about the variances. We use the "lattice" package instead of the standard `hist` function, as it is easier to keep the same scale on all three figures.

```
> y = numeric()
> for (i in 1:3) {
+ s = logical(3); s[i] = TRUE
+ y = c(y, dist.dna(cyn.ali[, s], p = TRUE))
+ }
> g = gl(3, length(y) / 3)
> library(lattice)
> histogram(~ y | g, breaks = 20)
```



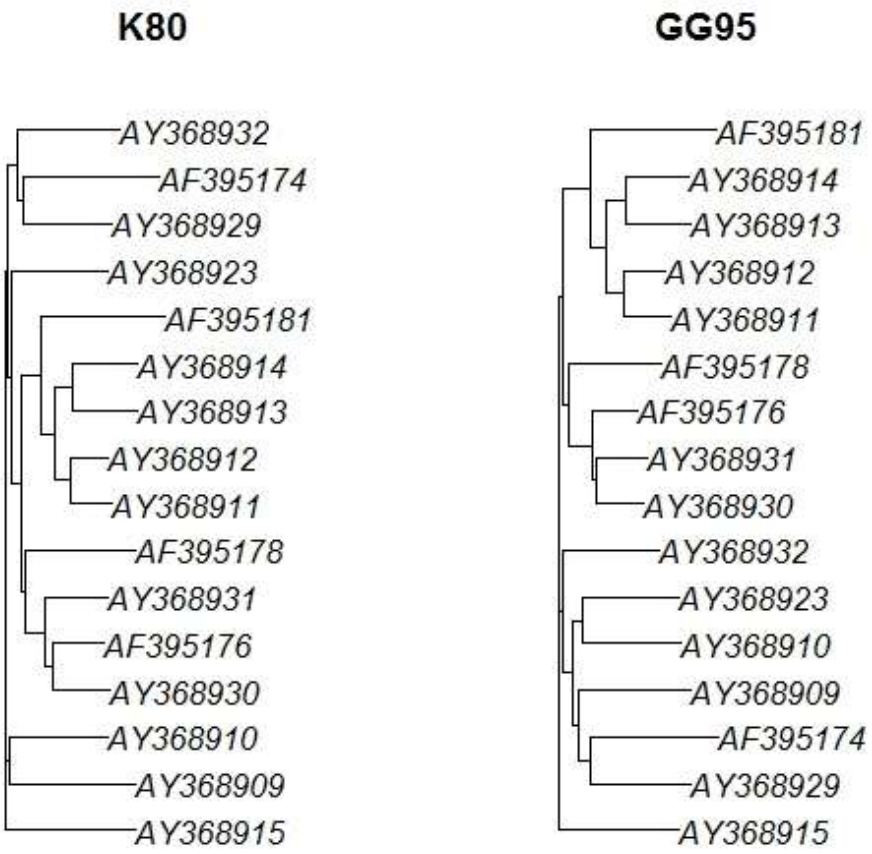
Phylogenetic reconstruction

Let us compare first the effect of model choice on the phylogenetic estimation. We use the neighbor-joining (NJ) method for illustration.

```
> nj.cyn.K80 = nj(cyn.K80)
> nj.cyn.GG95 = nj(cyn.GG95)
```

To have a first look, we can plot the trees easily using the default settings.

```
> layout(matrix(1:2, 1))
> plot(nj.cyn.K80,main="K80")
> plot(nj.cyn.GG95,main="GG95")
> layout(matrix(1,1))
```

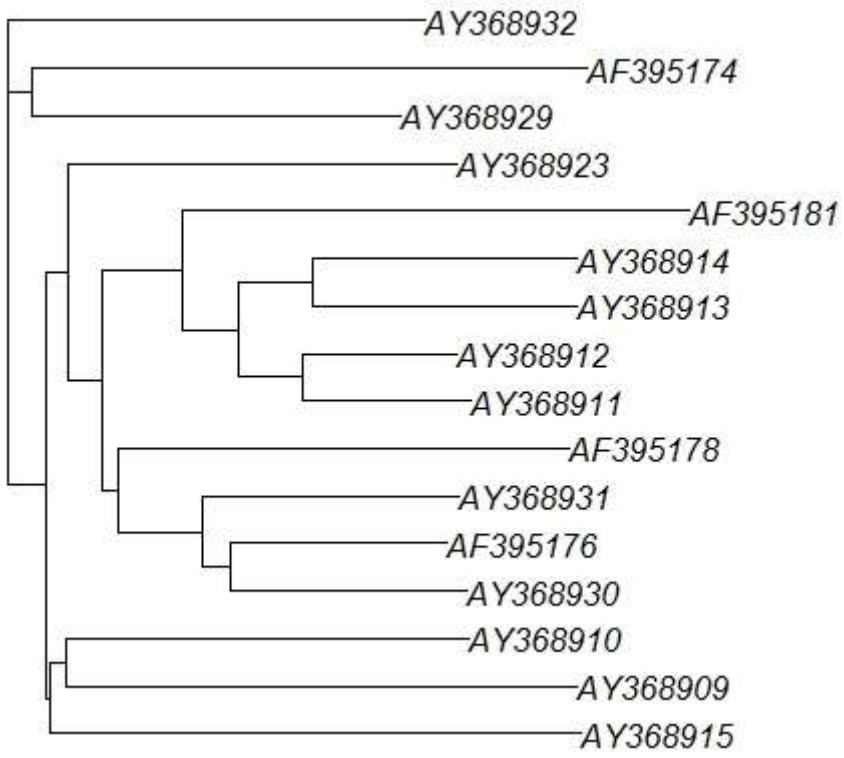


The difference between them can be measured by the topological distance.

```
> dist.topo(nj.cyn.K80, nj.cyn.GG95)
      tree1
tree2    10
```

The root of the tree can be specified by the `root` function. We have to give the outgroup as an argument of the function. The species *Parnips_nigripes* (accession number AY368932) was involved for this reason.

```
> plot(root(nj.cyn.K80, "AY368932"))
```



The next step is to perform bootstrap analysis. There are several ways for bootstrapping. We will use the function `boot.phylo` with rooted trees. First, we define a function for reconstruction.

```
> f = function(xx) root(nj(dist.dna(xx, p=TRUE)), "AY368932")
```

Next, we create the tree and store it in "tr". The "tr" object (it is a list) represents a phylogenetic tree with class "phylo".

```
> tr = f(cyn.ali)
> str(tr)
List of 4
 $ edge       : int [1:29, 1:2] 17 19 18 18 20 20 19 21 22 23 ...
 $ edge.length: num [1:29] 0.00685 0.000737 0.091836 0.00255 0.088756 ...
 $ tip.label  : chr [1:16] "AY368909" "AY368910" "AY368911" "AY368912" ...
 $ Nnode      : int 14
 - attr(*, "class")= chr "phylo"
 - attr(*, "order")= chr "cladewise"
```

The function `f` created above is called in each bootstrap replicate (200 times in this case).

```
> nj.boot.cyn = boot.phylo(tr, cyn.ali, f, 200, rooted = TRUE)

Running bootstraps:       200 / 200
Calculating bootstrap values... done.

> nj.boot.cyn
 [1] 200  16  93  61  62 115 106 110 198 199 143 192 186 185
```

The ith element of the latter vector is the number associated to the ith node of "tr" (we have 14, as it is given by `tr$Nnode`).

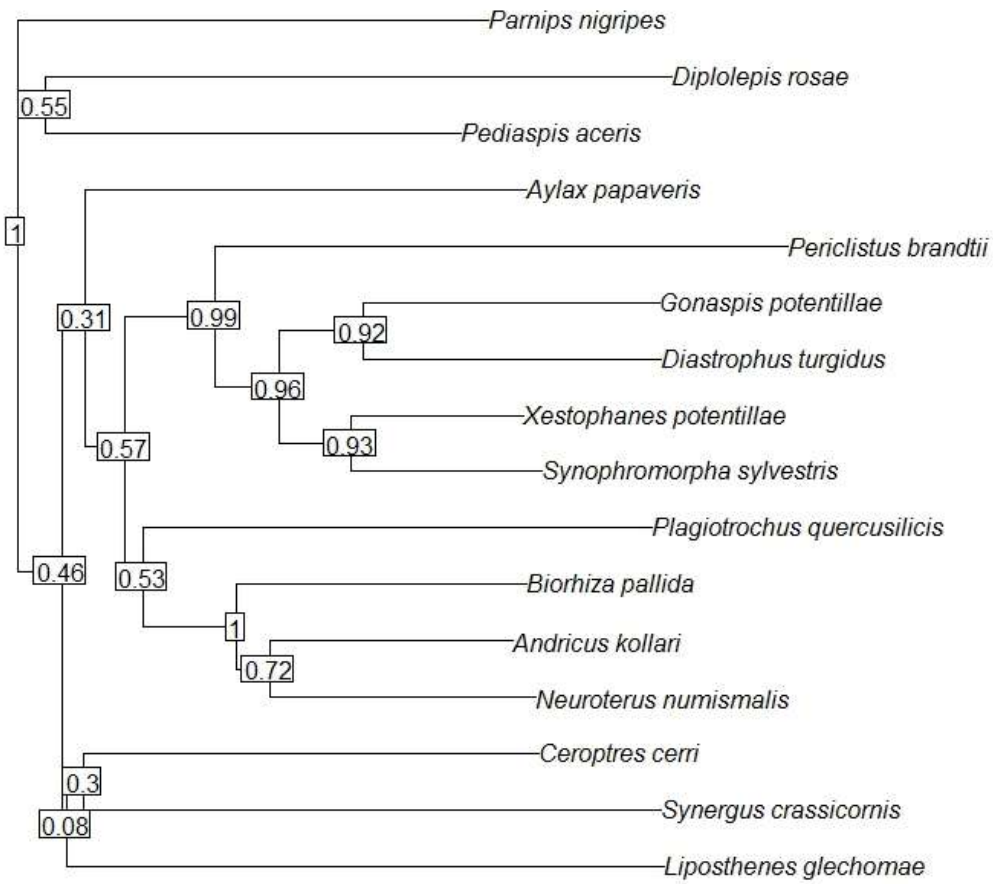
We are working with coding sequences, so codon level resampling may be better for our purposes.

```
> nj.boot.codon = boot.phylo(tr, cyn.ali, f, 200, block=3, rooted=TRUE)

Running bootstraps:       200 / 200
Calculating bootstrap values... done.
Warning message:...

> nj.boot.codon
 [1] 200  36 159  86  77 163 141 122 195 200 186 191 196 195
```

Note that we obtained a warning as the length of the sequence is not a muliple of 3.

We can now plot the estimated NJ tree with the bootstrap values on the nodes. Furthermore we substitute the accession numbers with species names from "taxa.cyn" (accession numbers are the names of the rows, so we can use them for reordering).

```
> nj.cyn = tr
> nj.cyn$tip.label = taxa.cyn[tr$tip.label]
> plot(nj.cyn, no.margin = TRUE)
> nodelabels(round(nj.boot.cyn/200, 2), bg = "white")
```



Scale bar can be added using `add.scale.bar`.

Finally, we can save the tree in Newick format:

```
> write.tree(nj.cyn, "cyn_nj_k80.tre")
```

Figures can also be saved from R keeping the vector format for zooming (postscript, pdf).

Phylogenetic reconstruction using maximum likelihood and model selection

Now we will use PhyML. It must be installed independently of R. We can call it from R using the `phymltest` function, but aligned sequences must be passed to PhyML in a file (it is already saved, check "cyn.txt" in the working directory, use `getwd()` if it is not clear). Running maximum likelihood analysis may be a time consuming task.

```
> phyml.cyn = phymltest("cyn.txt", execname = "phyml")
                        ........................
oooooooooooooooooooo    CURRENT SETTINGS       ooooooooooooooooooooooo
                        ........................

                . Sequence filename:                                cyn.txt
                . Data type:                                        dna
                . Alphabet size:                                    4
                 . Sequence format:                                  interleaved
                  . Number of data sets:                             1
                  . Nb of bootstrapped data sets:                    0
                  . Model name:                                      JC69
                  . Proportion of invariable sites:                  0.000000
                  . Number of subst. rate categs:                    1
                  . Optimise tree topology:                          yes
                  . Tree topology search:                            NNIs
                  . Starting tree:                                   BioNJ
                  . Add random input tree:                           no
                  . Optimise branch lengths:                         yes
                  . Optimise substitution model parameters:          no
                  . Run ID:                                          none
                  . Random seed:                                     1548608834
                  . Subtree patterns aliasing:                       no
                  . Version:                                         20120412

oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo

. 377 patterns found (out of a total of 1082 sites).
. 629 sites without polymorphism (58.13%).
. Computing pairwise distances...
. Building BioNJ tree...
...
```

Phymltest fits 28 models of DNA sequence evolution using PhyML and returns with the log-likelihood value of each model. PhyML results, including the trees, are saved into files in the working directory.

```
 > phyml.cyn
          nb.free.para    loglik      AIC
JC69                 1 -7718.556 15439.11
JC69+I               2 -7251.634 14507.27
JC69+G               2 -7164.107 14332.21
JC69+I+G             3 -7159.321 14324.64
K80                  2 -7718.534 15441.07
K80+I                3 -7249.740 14505.48
K80+G                3 -7164.057 14334.11
K80+I+G              4 -7159.313 14326.63
F81                  4 -7369.560 14747.12
F81+I                5 -6843.155 13696.31
F81+G                5 -6733.356 13476.71
F81+I+G              6 -6726.680 13465.36
F84                  5 -7356.862 14723.72
F84+I                6 -6815.458 13642.92
```

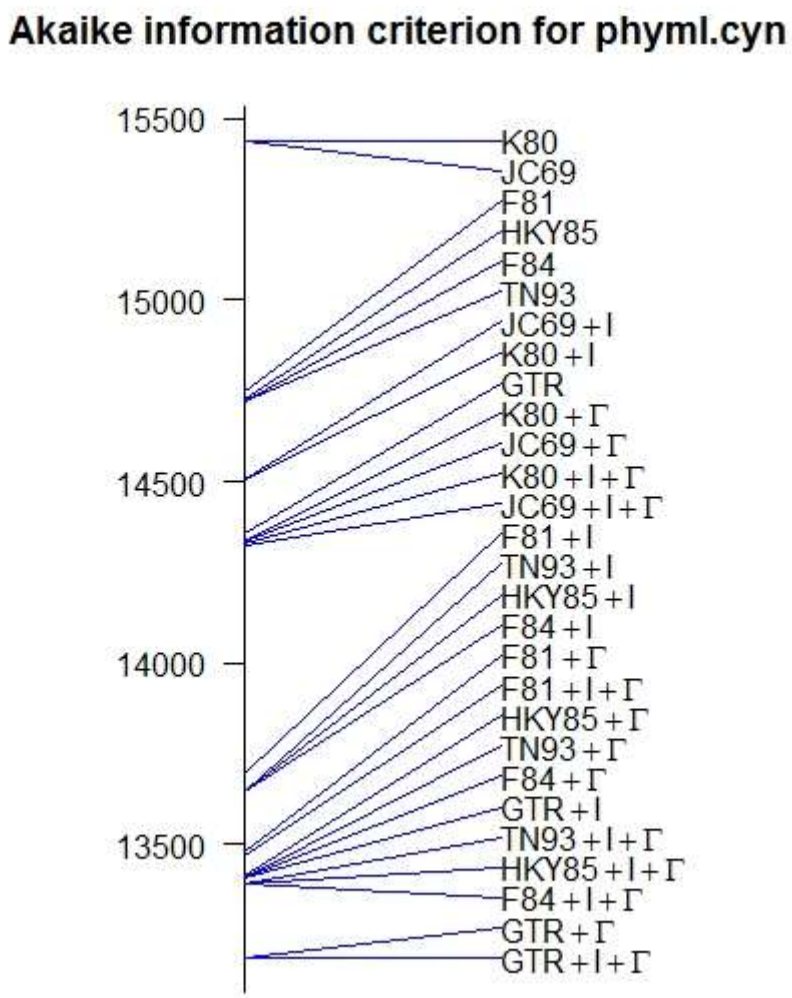
```
F84+G                6 -6697.210 13406.42
F84+I+G              7 -6688.444 13390.89
HKY85                5 -7357.745 14725.49
HKY85+I              6 -6815.689 13643.38
HKY85+G              6 -6699.365 13410.73
HKY85+I+G            7 -6688.819 13391.64
TN93                 6 -7355.162 14722.32
TN93+I               7 -6815.452 13644.90
TN93+G               7 -6697.193 13408.39
TN93+I+G             8 -6688.364 13392.73
GTR                  9 -7169.160 14356.32
GTR+I               10 -6692.657 13405.31
GTR+G               10 -6583.552 13187.10
GTR+I+G             11 -6581.451 13184.90
```

The paired likelihood ratio tests:

```
> summary(phyml.cyn)
       model1    model2       chi2 df  P.val
1        JC69    JC69+I  933.84370  1 0.0000
2        JC69    JC69+G 1108.89884  1 0.0000
3        JC69  JC69+I+G 1118.47100  2 0.0000
4        JC69       K80    0.04352  1 0.8347
...
```

Plotting the result may help to overview the fits.

```
> plot(phyml.cyn)
```



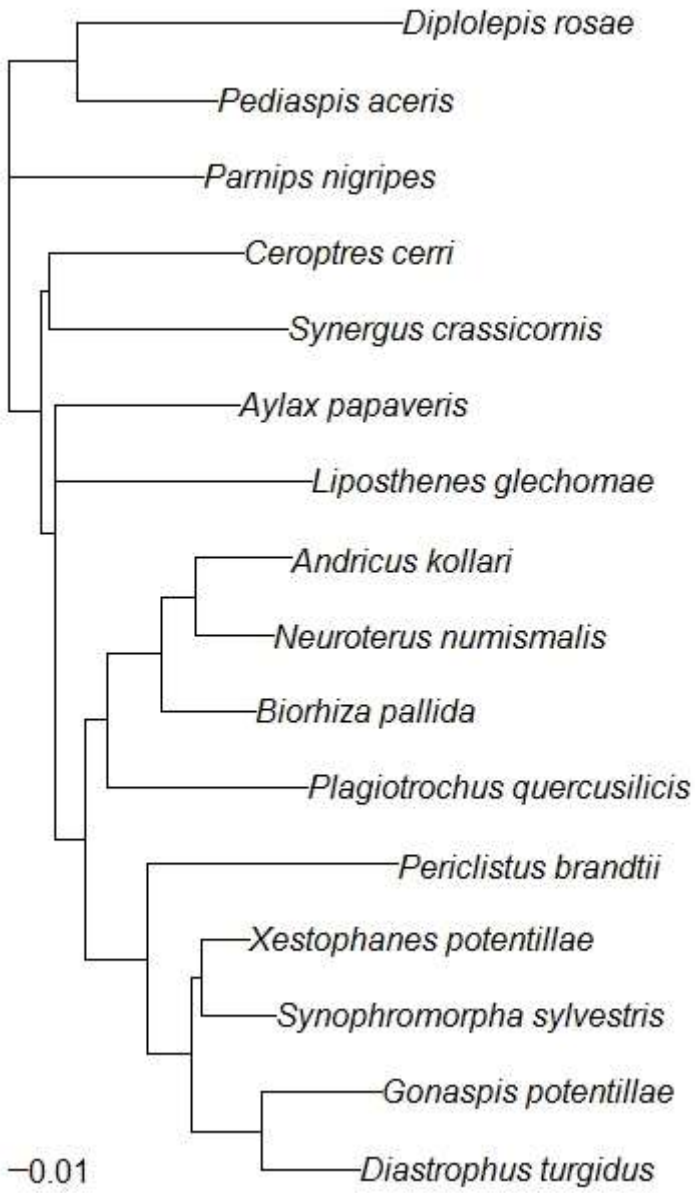
The most complex model, general time reversible model (GTR) with proportion of invariable sites (I) and gamma distributed rate variation among sites (Γ) is the best in terms of AIC. In general adding invariant proportions and gamma improves the fit of the models. We can also recognize other aspects (e.g. importance of transition/transversion ratio or unequal base frequencies).

PhyML saved the trees into the file "cyn.txt_phyml_tree.txt".

```
> tr = read.tree("cyn.txt_phyml_tree.txt")
> tr
28 phylogenetic trees
```

It contains 28 trees according to the 28 models. The most complex model, we selected recently, is the last one. We can extract and plot it:

```
> mltr.cyn = tr[[28]]
> mltr.cyn$tip.label = taxa.cyn[mltr.cyn$tip.label]
> mltr.cyn = root(mltr.cyn, "Parnips_nigripes")
> plot(mltr.cyn, no.margin = TRUE)
> add.scale.bar(length = 0.01)
```



To save the tree:

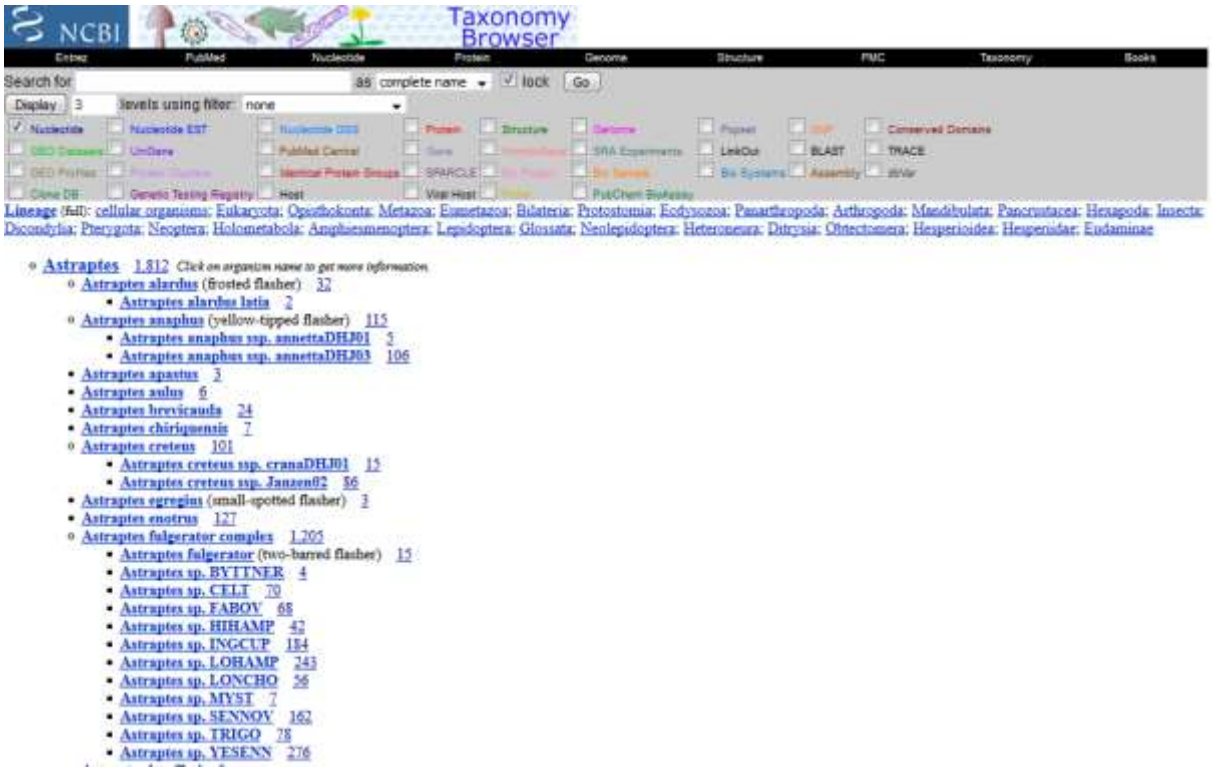
```
> write.tree(mltr.cyn, "cyn_ml_gtrig.tre")
```

The next step should be to fit a partitioned model (see `pmlPart` in the "phangorn" package). There are many more possibilities provided by the "ape" package (see Paradise 2011).

DNA barcodes

DNA barcoding approach is illustrated on a data set of a butterfly, *Astraptes fulgerator*, which is one of the classical examples for barcoding (reproduced from Paradise 2011).

GenBank contains many records for *Astraptes fulgerator* complex (check http://www.ncbi.nlm.nih.gov/Taxonomy/ and search for Astraptes).



We download 466 sequences from the GenBank, following Paradise (2011). Individuals were sequenced for a fragment of the mitochondrial gene cytochrome c oxydase I.

```
> x = paste("AY66", 6597:7060, sep = "")
> x = c(x, "AY724411", "AY724412")
> astr.seq = read.GenBank(x)
```

Let us check the distribution of the length of the sequences.

```
> table(sapply(astr.seq, length))
```

```
208 219 227 244 297 370 373 413 440 548 555 573 582 599 600 601 603 608
  1   1   1   1   1   1   1   1   1   1   3   1   1   1   1   2   2   1
609 616 619 620 623 626 627 628 629 630 631 632 633 634 635 636 638 639
  1   2   1   1   1   4   1   5   3   4   3   7   1   1  12   6   2 389
```

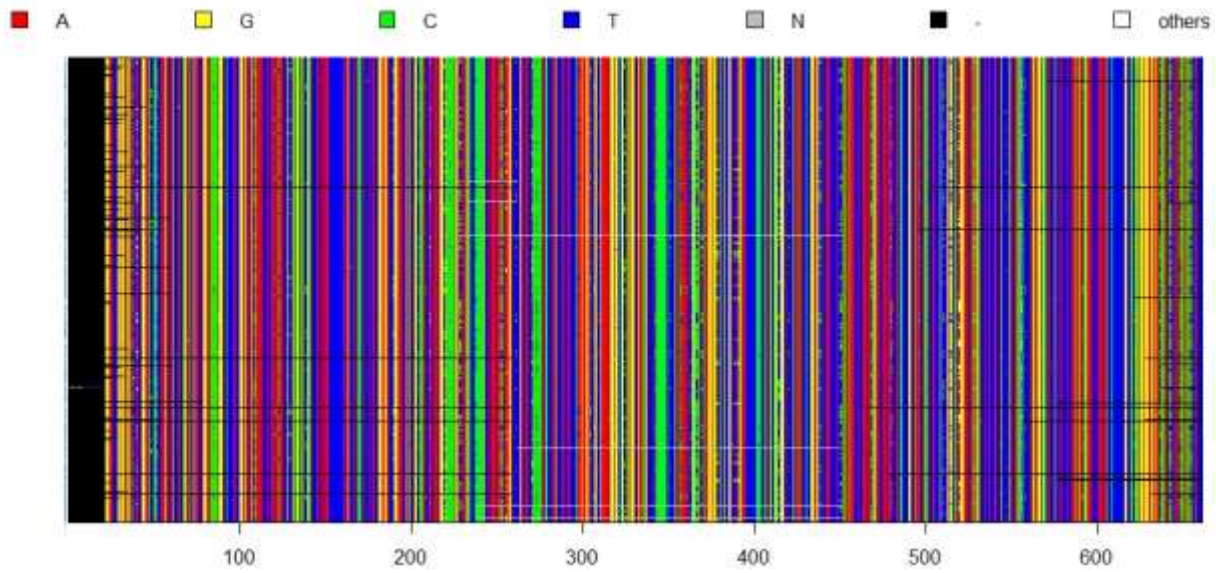
Lengths vary between 208 and 639, so we have to align them. However, the dataset is large, so the alignment using Clustal takes time.

```
> astr.ali = clustal(astr.seq)
```

Alternatively, save (`write.FASTA(astr.seq,"astraptes.fasta")`), align externally, and read the alignment back (`astr.ali = read.FASTA("astraptes_clustal.fasta")`).

Let us see the alignment (the "barcode" of the samples):

```
> image(astr.ali)
```



We check the species names and store them together with the accession numbers:

```
> table(attr(astr.seq, "species"))

Astraptes_sp._BYTTNER    Astraptes_sp._CELT   Astraptes_sp._FABOV
                    4                    23                    31
 Astraptes_sp._HIHAMP  Astraptes_sp._INGCUP  Astraptes_sp._LOHAMP
                   16                    66                    47
 Astraptes_sp._LONCHO    Astraptes_sp._MYST  Astraptes_sp._SENNOV
                   41                     3                   105
  Astraptes_sp._TRIGO  Astraptes_sp._YESENN
                   51                    79
> taxa.astr = attr(astr.seq, "species")
> names(taxa.astr) = names(astr.seq)
```

Identifiers are given only for specimens; species names are unknown. Then, as before, save the data for later use:

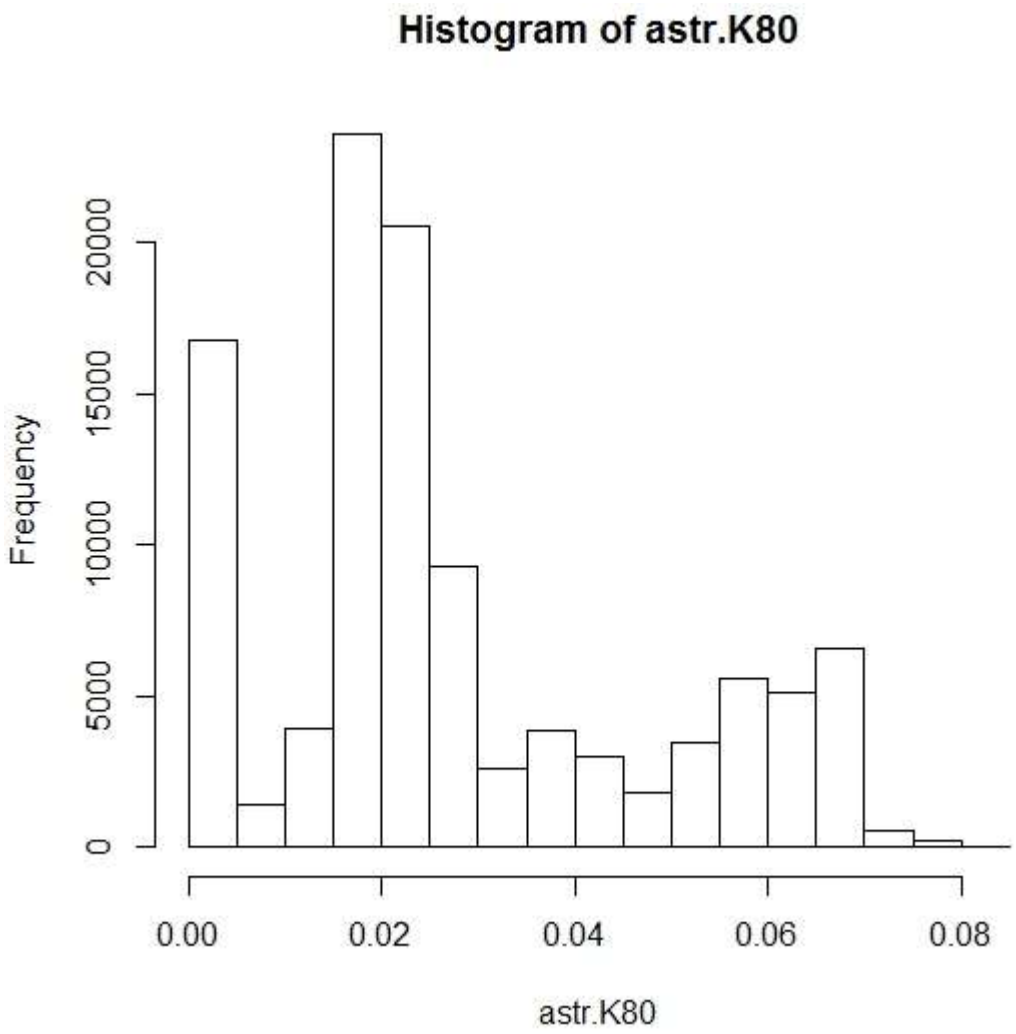
```
> save(astr.ali, taxa.astr, file = "astraptes.RData")
```

From these data researchers suggested that there are about 10 species instead of one originally recognized species. Let us see the pairwise genetic distances. Because of the length differences, we must take care of pairwise deletion. DNA barcoding uses Kimura's two parameter (K80) model traditionally, which is the default for `dist.dna`.

```
> astr.K80 = dist.dna(astr.ali, pairwise.deletion = TRUE)
```

Let us look at the distribution of distances (we have 108345, `length(astr.K80)`)

```
> summary(astr.K80)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.00000 0.01590 0.02101 0.02747 0.03887 0.08326

> hist(astr.K80)
```



There are peaks at 0, around 0.02 and around 0.07. It was hypothesized that this pattern of distances correspond to the within population, among populations and among species differentiation.

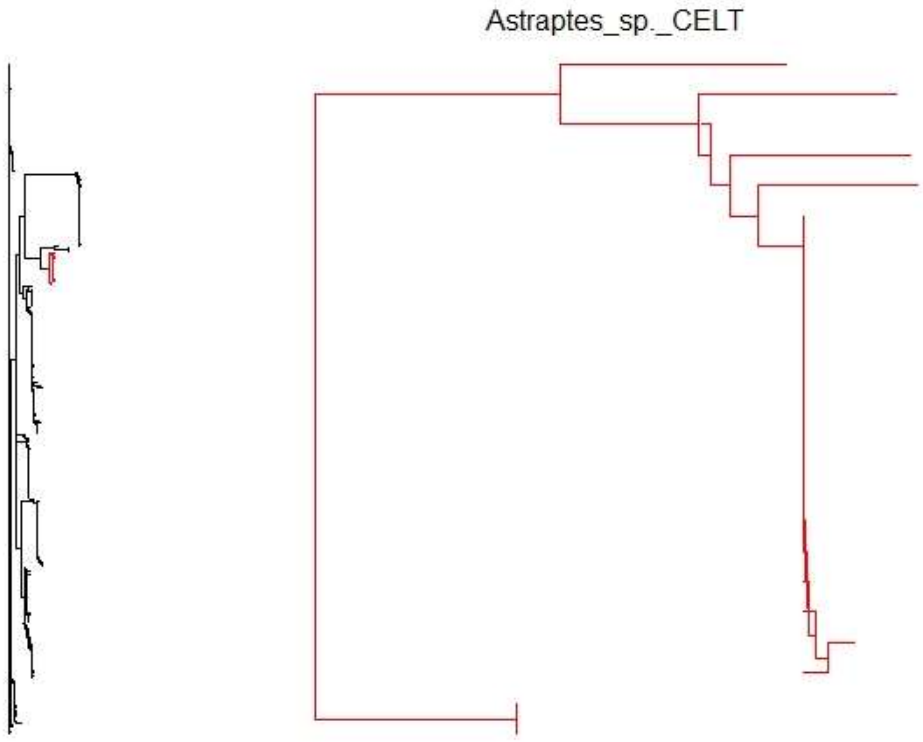
Let us visualize this distance matrix using NJ.

```
> tr = nj(astr.K80)
> tr$tip.label = taxa.astr[tr$tip.label]
```

However, the tree is large to be plotted in the usual manner, even without tip labels (try `plot(tr,type="unrooted",show.tip.label=FALSE)`).

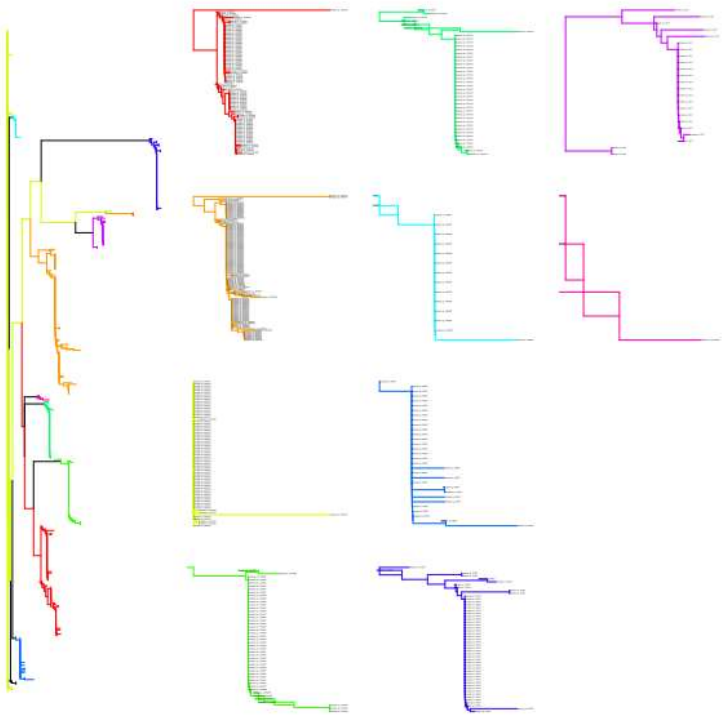
Ape package provides a solution using the function `zoom` instead. We may collect the indices of a given taxon and zoom the subtree(s). For example (from Paradise 2011):

```
> tx = unique(taxa.astr)
> L = vector(mode = "list", length = 10)
> for (i in 1:10) L[[i]] = grep(tx[i], tr$tip.label)
> zoom(tr,L[[9]], show.tip.label=F); mtext(tr$tip.label[L[[9]][1]])
```



Finally, we can plot all subtrees into a large PDF file, where colours denote different lineages according to the L index.

```
> pdf("astraptes.pdf", width = 30, height = 30)
> zoom(tr, L)
> dev.off()
```



References

Nylander, JAA et al (2004) Bayesian Phylogenetic Analysis of Combined Data. Systematic Biology 53:47-67

Paradis E (2011) Analysis of Phylogenetics and Evolution with R (2nd ed.). Springer

SUMMARY

R is a usefull platform for many stages of the phylogenetic analyses and DNA barcoding.

R functions used in Chapter 12

abline
add.scale.bar
as.character
attr
boot.phylo
cbind
cor
clustal
dev.off
dist.dna
dist.topo
for
function
getwd
gl
grep
head
hist, histogram (lattice)
image
is.na
layout
library
load
logical
match
matrix
mtext
names
nj
nodelabels
numeric
paste
pdf
phymltest
points
read.GenBank
read.FASTA
read.table
read.tree
rm
root
round
rownames
sapply
save
str
summary
table
unique
write.dna
write.FASTA
write.tree
zoom

Description of the subject

Master level

Title: Biostatistics	**Credits:** 3
Category of the subject: compulsory	
The ratio of the theoretical and practical character of the subject: 10-90 (credit%)	
The type of the course: laboratory practice **The total number of the contact hours**: 2 per week **Language***: English*	
Evaluation: solving tasks, written **Other methods for evaluation of the student's competence** *(if any)***:**	
The term of the course: I. semester	
Prerequisites *(if any)*: -	

Description of the subject

Aims:

The course is designed to provide the students with the basic mathematical and statistical methods useful for biologists. R is a free software environment for statistics and graphics. The aim of this course is to learn R via a knowledge of basic statistics. The course surveys the most important methods as implemented in R.

Students become able to use R to summarize and graph data, calculate confidence intervals, test hypotheses, assess the goodness-of-fit, and perform analyses of variance and linear regression. They also acquire a basic knowledge of the methodology behind modern data-based modelling (simulations, maximum likelihood, bootstrapping and Bayesian analysis).

Topics of the course:

- Basics of R.
- Descriptive statistics and graphics.
- One-sample tests. Two-sample tests.
- Regression and correlation.
- Analysis of variance and nonparametric alternatives.
- Categorical data.
- Power analysis, simulations in R.
- Linear models.
- Nonlinear models.
- Multivariate analysis.

Selected bibliography (**2-5)** (author, title, edition, ISBN)

Dalgaard P: Introductory Statistics with R. 2nd ed., Springer Science+Business Media, LLC, New York, 2008. ISBN: 978-0-387-79053-4

R software documentation (http://www.r-project.org/)

General competence *(knowledge, skills,* etc., *KKK **8.***) **promoted by the subject**

a) knowledge

Students will

- be familiar with the basic methods of modern biology.
- know the terminology of modern biology and apply it in the correct way.
- understand the social problems with biological relevance.
- understand the significance of the interdisciplinary approach.
- know the coherency among the area of biology.

b) skills

Students will be able to

- participate in biological research project and to create new scientific results under competent supervision.
- plan research projects in the field of biology
- recognize the coherency among the different area of biology.
- apply new methods and techniques independently.
- interprete and the present their results in a correct way.

c) attitude

Students will be

- open to cooperate with other research groups
- ready to understand the evolution, structure and function of the living organisms
- interested in new results, techniques and methods; contribute to new scientific results and methods
- keep the ethical rules of the biological research

d) autonomy and responsibility

Students will

- be able to organize the work of small research teams independently
- help his collegues in the completion of the research projects
- build their own scientific career consciously

Special competence promoted by the subject:

Knowledge	Skills	Attitude	Autonomy/responsibility
Students will know the basic statistical methods useful in biology.	Students will be able to use R to evaluate their data.	Open to study new methods. Ready to prepare a correct notebook. Ready to cooperate with his/her colleagues in the completion of the experiments.	Apply the acquired methods independently. Evaluate the results independently and correctly.
Students will know the basic functions of R.	Students will be able to use R to summarize and graph their data.		
Students will know the methods to test a hypothesis.	They will be able to test hypothesis.		
Student will know the logic of the analysis of variance and linear regression.	They will be able to perform analyses of variance and linear regression among others.		
Students will know the methodology behind modern data-based modelling	They will be able to perform simulations.		

Instructor of the course (*name, position, scientific degree*)**:**

Dr. Pénzes Zsolt, associate professor, PhD

Teachers (*name, position, scientific degree*)**:**

Dr. Pénzes Zsolt, associate professor, PhD

Dr. Tölgyesi Csaba, assistant professor, PhD

This teaching material has been made at the University of Szeged, and supported by the European Union. Project identity number: EFOP-3.4.3-16-2016-00014

University of Szeged, Hungary
Venue: H-6720 Szeged, Dugonics square 13.
www.u-szeged.hu
www.szechenyi2020.hu