How-to in R

Wiebe R. Pestman

Contents

1 Histograms

How to make a histogram in R? In this section you will learn how to do this. To have some data to work with, create a sequence `height` of 50 numbers by passing the following command (this command is be explained in the section Generating random data):

```
height <- rnorm(50,175,10)
```

and strike the key `Enter`. It will seem as if nothing has happened. Pass the command

```
height
```

and strike the key `Enter`. You will then see a list of 50 numbers indeed: You have been storing the sequence of 50 numbers in an object `height` that resides in the so-called *workspace* of R (see the section Numbers, strings and objects in R). Pass the command

```
ls()
```

or

```
objects()
```

to see a complete list of all the objects in your workspace.
To create a histogram of the sequence `height`, just pass the command

```
hist(height)
```

and strike the key `Enter`. A graphics window will then appear with in it a histogram presentation of the sequence `height`.

To see the extensive possibilities of the histogram command, just type

```
?hist
```

Play around a bit with this command. For example, change the colour of the histogram and the number of intervals. Use the option `xlab="blablabla"` to add a label to the horizontal axis. By using `ylab` istead of `xlab` you may add labels to the vertical axis. See the section Plotting curves to learn how a normal curve can be displayed in the histogram.

If you want to remove the object `height` from the workspace, enter the command

```
rm(height)
```

The object `height` is then removed from your workspace. To check this, just pass the command

```
ls()
```

as you did earlier.

2 Bar plots and pin diagrams

How to make a bar plot in R? As an example, let us take the table below:

score	frequency
1	71
2	113
3	83
4	134
5	103
6	96

Presenting this table graphically as a bar plot in R can be done as follows: First put in an object, say `freq`, like this

```
freq <- c(71,113,83,134,103,96)
```

Then pass the command

```
barplot(freq)
```

and your bar plot will appear. In this bar plot, however, the scores are not indicated on the horizontal axis. To bring this about, do the following. First make an object, say `scores` that contains the scores.

```
spots <- c(1,2,3,4,5,6)
```

Then weight the scores with the frequencies stored in the object `freq` like this

```
john <- rep(spots,freq)
```

Print the object `john` to the screen to see its contents. Run the command

```
table(john)
```

Finally, run the command

```
barplot(table(john))
```

This will provide the desired bar plot.

The table can also be presented as a pin diagram. To this end, pass the command

```
plot(spots,freq)
```

You will see a figure appear, but it is not the type of figure you wish to see. Close the figure and repeat the above with the option to specify a type

```
plot(spots,freq,type="h")
```

You will then get your desired pin diagram. Just for fun, also try

```
plot(spots,freq,type="l")
```

Here `"l"` stands for the 12th character in the alphabet; it is *not* the number 1. The graphic utility `plot()` in R is very versatile. You can do a lot of graphic work with it. Type

```
?plot
```

to get an impression of the possibilities of this utility.

3 Cumulative frequency plots

How to make a cumulative frequency plot in R? To illustrate this, produce some data to work with and store it in the object `x` (for example the flood size data of the Ocmulgee River)

```
x <- c(50,12,16,20,17,13,61,26,33,38)
```

Next, sort the values in `x` from low to high

```
x <- sort(x)
```

Now store the length of the sequence `x` in an object `n`:

```
n <- length(x)
```

So, in this example, `n` has the value 10. Define the cumulative frequencies as

```
y <- (1:n)/n
```

Print the values of `y` to the screen to see what you have been doing by the above.

Next, the graphic utility `plot()` can be used to produce the cumulative frequency plot:

```
plot(x,y,type="s")
```

If you want to include labels on the x-axis and y-axis, then do something like

```
plot(x,y,type="s",xlab="Size of Flood")
```

Similarly a label can be attached to the y-axis. Don't forget the quotation marks!

Alternatively, probably easier, a cumulative frequency plot can be produced by means of the function `ecdf()`. It is not necessary to sort the sequence `x` then. Just run the command

```
john <- ecdf(x)
```

By this action the empirical distribution function is created and named `john`. If you type the command `john(27)` then you get the value 0.6 returned. This indicates that 60% of the data in `x` is less than 27. A plot can be obtained by passing a command of type

```
plot(john,verticals=TRUE,do.points=FALSE)
```

Also try this without the options `vericals=TRUE` and `do.points=FALSE`.

4 Saving graphics

The graphics produced in an R-session can of course be saved to files. The function `dev.copy()`, does the job in this. You can save a given plot on your screen as follows:

```
dev.copy(jpeg,file="X:/path/to/mygraph.jpg")
```

Here the letter `X` stands for the drive on which you want to have your file saved. By the above a file `mygraph.jpg` is created and a bitstream towards this file is started. To close the stream, pass the command

```
dev.off()
```

Your figure is now saved in a jpg file `X:/path/to/mygraph.jpg`. If you simply type

```
dev.copy(jpeg,file="mygraph.jpg")
dev.off()
```

then the file `mygraph.jpg` is saved in the working directory. Several options are available to tune the process. For example, one can set the width and height of the picture (in pixels) as follows:

```
dev.copy(jpeg,file="X:/path/to/mygraph.jpg",width=200,height=250)
dev.off()
```

Don't forget to close the bitstream by passing the command `dev.off()`. If you don't, it could result in misshapen files. Besides width and height, one could also set the background color and the orientation of the figure as follows:

```
dev.copy(jpeg,file="/path/to/mygraph.jpg",width=200,height=250,
                                           bg="red",horizontal=F)
dev.off()
```

In a similar way one can save graphics in the formats png, pdf and eps. In eps the command would be something like:

```
dev.copy(postscript,file="mygraph.eps",width=5,height=5)
dev.off()
```

Rather than in pixels, in postscript (eps) the width and height of the figure must be put in in inches!

Note that, when running R on a Windows machine, you can also use the function `savePlot()` to save your graphics. Pass the command `?savePlot` to get information about this method.

5 Numbers, strings and objects in R

When working in R it is comfortable to know a little bit about bits, bytes, strings, objects etcetera. This section is a quick guide to these kind of things. The explanation will be based on similarity to things that you have learnt at school during chemistry lessons. In one of your first chemistry lessons you probably were taught that *atoms* are thought to be the building blocks in this science. By means of these atoms one can build molecules. Molecules, in turn can be ranked in molecular arrays, such as lattices, chains, double helices etcetera.

In computer sciences the atoms can be thought to be *characters* and *digits*. Examples of characters are the symbols "a", "B", "!", "y", "@", ";" etcetera. A space, that is " ", is a less trivial example of a character. Digits are understood to be the symbols 0, 1, 2, . . . , 8, 9. Digits distinguish themselves from characters in that they allow for operations like multiplication, addition and so on (see footnote). Characters fail to have these properties. Thus the atoms of informatics come in two flavors: characters and digits. Note that in chemistry atoms also come in flavors: metals, inert gases, halogens, . . .

The molecules in informatics are *numbers* and *strings*. One can bind together digits to form numbers, for example 112, 1001, 47, 0.37, ... Characters can be compounded into so-called

A highly confusing thing is here that the symbols 0, 1, 2, . . . , 9 can also be used as being characters, that is, as characters "0", "1", "2", . . . , "8", "9". They just serve as symbols then; so one cannot multiply the character "3" by the character "2".

strings, for example "Avenue Saint Laurent; Montreal". Note that the latter molecule (or string) contains spaces among its atoms.

In chemistry the molecules often build chains, lattices, cycles etcetera. In informatics, specifically in R, the molecules can be used (or stored) in so-called *objects*. Rather than chains, crystals, cycles, etcetera, the objects in R are called *vectors*, *matrices*, *data frames*, *lists* and *factors*. Below you can find a brief description of these types of objects.

vectors A *vector* in R could be thought to be the equivalent of a chain of molecules in chemistry. Vectors are sequences of numbers and/or strings. Vectorial objects can be regarded as being the most simple objects that R offers the user. To give an example, with the three molecules or strings "john", "peter" and "nicholas" one could build a vector (chain, sequence) in R, named `myfirstvector`, like this

```
myfirstvector <- c("john","peter","nicholas")
```

By this action R creates an object, named `myfirstvector`, that contains the strings "john", "peter" and "nicholas" as its basic components. The components of the vector `myfirstvector` can be displayed on the screen by simply typing `myfirstvector` and striking the key `Enter`. Order matters! The vector defined by

```
mysecondvector <- c("peter","nicholas","john")
```

differs from `myfirstvector`! A vectorial object may also contain just numbers; try the following in R:

```
mythirdvector <- c(47,21,10,9)
```

A component, say the 4th component of the vector `mythirdvector` can be accessed by typing

```
mythirdvector[4]
```

Components can be overwritten like this:

```
mythirdvector[4] <- 11
mythirdvector[2] <- "jane"
```

The 3th component can be deleted from the vector by typing

```
mythirdvector[-3]
```

If the component's of a vector are all numerical, then one may apply arithmetic operations to it. For example, one may evaluate

```
3*mythirdvector
```

Last but not least, a vectorial object in R may contain both numbers and strings:

```
myfourthvector <- c(23,"john",10,9)
```

Arithmetical operations are forbidden for vectors hat contain strings among their components.

matrices A matrix in R could be thought to be the equivalent to a 2-dimensional (or flat) rectangular lattice in chemistry. A matrix has columns and rows. To see an example in R, first make a vector with 6 components:

```
myvector <- c(2,1,9,3,3,4)
```

Now turn this vector into a matrix with 3 rows and 2 columns, that is a 3x2-matrix, as follows:

```
myfirstmatrix <- matrix(myvector,ncol=2)
```

Print the contents of `myfirstmatrix` to the screen to see what you created through the above. The matrix `myfirstmatrix` could also have been created via the command

```
myfirstmatrix <- matrix(myvector,nrow=3)
```

The element on row number 3 and column number 2 can be accessed like this

```
myfirstmatrix[3,2]
```

Row number 3 and column number 2 can be accessed as follows:

```
myfirstmatrix[3,]
myfirstmatrix[,2]
```

Row number 3 can be deleted from the matrix by running

```
myfirstmatrix[-3,]
```

In a similar way a column can be deleted. To transpose a matrix, that is to say, to turn the rows into columns (and the columns into rows) do the following:

```
mysecondmatrix <- t(myfirstmatrix)
```

The elements of a matrix are either numbers or strings; it is not possible to store both strings and numbers in one and the same matrix. Numerical matrices can be subjected to multiplication and addition, as defined in linear algebra. For example, the matrices `myfirstmatrix` and `mysecondmatrix` can be multiplied through the command

```
mythirdmatrix <- myfirstmatrix %*% mysecondmatrix
```

Thus the power of linear algebra can be exploited.

The rows of the matrix `myfirstmatrix` can be assigned names as follows

```
rownames(myfirstmatrix) <- c("john","peter","nicholas")
```

The columns can be equipped with headers like this:

```
colnamesnames(myfirstmatrix) <- c("mary","jane")
```

Print the contents of `myfirstmatrix` to the screen to see the effect of the above.

arrays An array is a generalized matrix. Generalized in the sense that the matrix is allowed to be of multi-dimensional size. A 2-dimensional array is just a matrix. A 3-dimensional array could be compared to a 3-dimensional cubic lattice of molecules. Alternatively it could be compared to a book the pages of which consist of matrices. As an example, given any vector `john` of length 120, it can be turned into a 3-dimensional array `peter` of size 4x5x6 like this

```
peter <- array(john,dim=c(4,5,6))
```

Now the object `peter` can be regarded as a book with 6 pages. Each page consists of a 4x5-matrix. The entry on row 3 and column 2 of the matrix on page 5 can be accessed by typing

```
peter[2,3,5]
```

Page 3 of the book `peter` can be accessed as

```
peter[,,3]
```

The same array could, of course, also be considered as a book with 3 pages where each page consists of a 5x6-matrix. Page 2 could then be accessed as

```
peter[2,,]
```

Arrays of higher dimensions can also be created. For example, one could create a 4-dimensional array `nicholas` of size 2x2x5x6 as follows:

```
nicholas <- array(john,dim=c(2,2,5,6))
```

The object `nicholas` could be compared to a library that contains 6 books. Each book is an array of size $(2, 2, 5)$.

lists In a list you can store virtually all kinds of things. To illustrate this, create three objects in R like this

```
x <- c("john","peter","nicholas")
y <- c("mary","jane")
z <- matrix(1:12,ncol=3)
```

These three objects can be stored in a list in the following way:

```
myfirstlist <- list(x,y,z)
```

Now the object `myfirstlist` contains the objects `x`, `y` and `z` as elements. You can access an object, say the object `y`, from the list by typing

```
myfirstlist[[2]]
```

The components of the list can be equipped with headers, say the headers `garcons`, `filles` and `mat`, by redefining it as

```
myfirstlist <- list(boys=x,girls=y,matrix=z)
```

The three components of the list can then also be accessed by typing

```
myfirstlist$boys
myfirstlist$girls
myfirstlist$matrix
```

A list may even contain lists as components:

```
mysecondlist <- list("hillary",myfirstlist)
```

In this list the object `x` can be accessed by typing

```
mysecondlist[[2]][[1]]
```

or, by using the headers in `myfirstlist`, by typing

```
mysecondlist[[2]]$boys
```

data frames Data frames resemble matrices in that they have columns and rows, but under the hood they are lists. They can be accessed as if they were matrices. Contrary to a matrix, however, a data frame may contain both numbers and strings. The price to be paid is that one cannot apply linear algebraic operations to data frames. For example, two data frames cannot be multiplied. Data frames are very popular storage objects among R users. This type of objects approaches the concept of a spreadsheet.

factors Factors resemble vectors, but under the hood they differ from vectors. If the purpose of the components of a vector is to define categories, that is, if the vector is used as a grouping variable, then it may sometimes be necessary to convert a vector into a factor. This is, for example, the case if you want to use a vectorial grouping variable as an explanatory variable in a regression analysis.

Given an arbitrary object in R, say the object `john`, you can ask R whether it is a vector, matrix, data frame, list or factor by running commands of the form

```
is.vector(john)
is.matrix(john)
is.data.frame(john)
is.list(john)
is.array(john)
is.factor(john)
```

Sometimes an object can be coerced into an object of another type. For example, a matrix, say `john`, can always be coerced into a data frame like this:

```
peter <- as.data.frame(john)
```

The object `peter` now contains the same data as `john`, but it is a data frame rather than a matrix. Similarly one has the coercion operations `as.vector`, `as.matrix`, ... Be aware that coercion is not always possible!

6 Writing data to plain text files

In R data can be saved in two ways: one can save data in plain text files and one can save data in a native (binary) format. Both methods have their specific advantages. If you want to save the content of, say, an object `height` in a plain text file named `foo.txt`, then run the following command

```
write(height,file="foo.txt")
```

By this a plain text file `foo.txt` is created in the working directory of R. This file can be read by text editors. By default the data in this file is in arranged in 5 columns. If you want to have your number material arranged in, say, 2 columns, then run

```
write(height,file="foo.txt",ncolumns=2)
```

Again the file will be stored in the working directory of R. If you don't know where this directory resides, then run

```
getwd()
```

to see the path to this folder on your screen.

If you want to have the text file saved on a place outside the working directory, then run something like

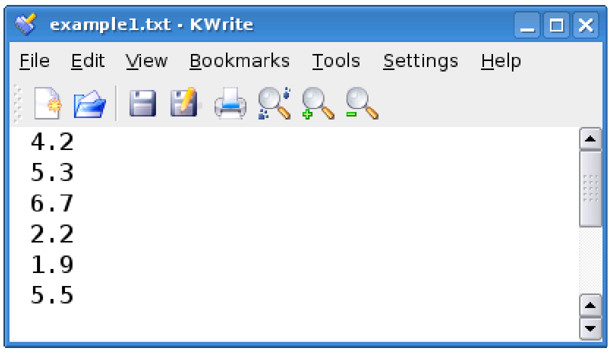
```
write(height,file="C:/path/to/foo.txt")
```

The advantage of saving an object in a text file is that these files can be read in not only to R, but also to other programs, such as SPSS, Matlab, Scilab, Stata, Excel etcetera. These other programs, like R, are also able to write data to text files. Thus text files present a powerful way to move data around over different programs. One of the disadvantages of saving data in text files is that one can save only simple objects in this way and only one per file.

In the next section it is explained how to read in data from text files to R.

7 Reading in data that is stored in plain text files

Suppose your data is stored in a plain text file, named `example1.txt`. The data could be as sketched in the figure below:



```
4.2
5.3
6.7
2.2
1.9
5.5
```

How to get this in your workspace in R? This can be done by means of the utility `read.table()`. First of all determine the path to the place where your file resides; this will be something of the form:

```
C:/path/to/example1.txt
```

The data can be retrieved from this place by passing the command:

```
read.table(file="C:/path/to/example1.txt")
```

Don't forget the quotations here! You will see the data directly printed to the screen. If you want to have your data stored in an object named, say `mystuff`, then pass the following command:

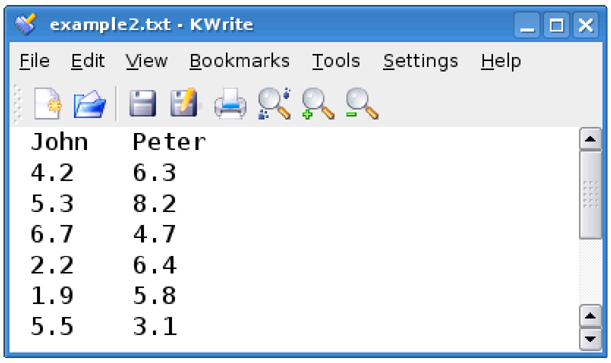
```
mystuff <- read.table(file="C:/path/to/example1.txt")
```

The data is now in your workspace and is stored in an object named `mystuff`.

The method sketched above also works if you want to retrieve data from your USB pendrive. Just find out which drive letter Windows assigned to your pendrive and then determine the right path to your file.

Be aware that the above only works for plain text files and *not* for MS-Word files in document format!

Next, suppose your data is stored in two columns in a text file, named `example2.txt`. The data could be as sketched in the figure below:



```
John    Peter
4.2     6.3
5.3     8.2
6.7     4.7
2.2     6.4
1.9     5.8
5.5     3.1
```

How to get this in your workspace in R in a 2-column layout with the headers included? Again you can use the utility `read.table()` to bring this about. Suppose that the path to the file `example2.txt` is

```
C:/path/to/example2.txt
```

Then the data can be retrieved from this place by passing the following command:

```
mystuff <- read.table(file="C:/path/to/example2.txt",header=TRUE)
```

The data is now in your workspace and is stored in an object named `mystuff`. Note that you can view the first and last six rows of `mystuff` by running the commands

```
head(mystuff)
tail(mystuff)
```

When running into problems during read-in, always check the following:

- Did you type in the right path to your text file?
- Did you use the right slashes in the path to your data file?
- Did you put quotations around the path?
- Did you use the right decimal separators in your dataset?
- If not a space or tab, did you specify the column separator?

8 Saving data to native file format

To illustrate how to save data to the R-native format, suppose you have the objects `height` and `bmi` in your workspace. Then you can save them in one file, say, the file `foo.RData` like this

```
save(height,bmi,file="foo.RData")
```

The file `foo.RData` can now be found in the working directory of R. The file contains the content of the two objects `height` and `bmi`, but the content cannot be read in a text editor. It is a so-called *binary* file and its format was specifically designed for usage in R.

In the next section it is explained how data can be loaded from native file format.

9 Loading data that is stored in native file format

Suppose your data is stored in objects that are stored in an R-file named `foo.RData`. How to get these objects in R? This is very easy; just pass the command

```
load(file="X:/path/to/foo.RData")
```

where `X` stands for the drive on which the file resides. All the objects saved in the file `foo.RData` are then in your workspace. Pass the command `ls()` to see what you got in your workspace. If everything went well you can view the first six rows of an object by using the function `head()`. More precisely, if you want to view the first six rows of an object named, say `height`, then run the following command

```
head(height)
```

Similarly you can use the function `tail()` to view the last six rows of an object.

10 Loading built-in data into your workspace

In R there are many of real datasets available. They have been built in during the installation of the program. To get an impression, just type

```
data()
```

Among the list that will appear on the screen there is for example a dataset named ‘faithful’. To get this dataset in your workspace, type

```
data(faithful)
```

Then, for example, type the commands

```
head(faithful)
tail(faithful)
```

to view respectively the first and last six rows of the dataset in question.

Of course you would like to know what the dataset is about. To get a description of the meaning of the columns in the dataset, pass the following command

```
help(faithful)
```

You will then learn that it is about eruption durations and waiting times as to some geyser in the Yellow Stone Park in the USA. Note that, when running R under Unix systems, you can quit the help screen by typing

```
q
```

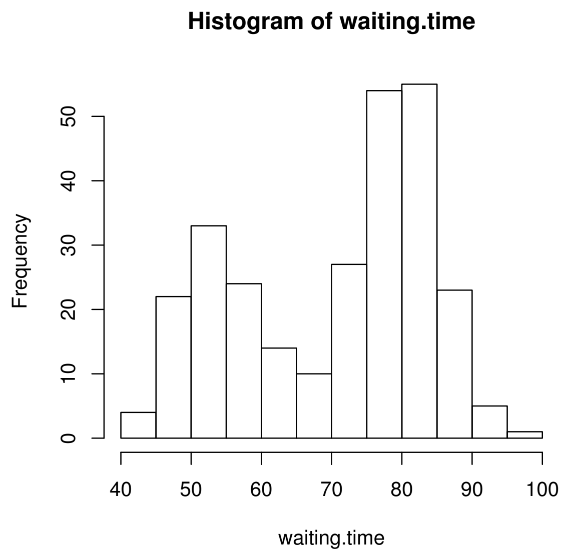
As a simple exercise one could make, for example, a histogram of the waiting times like this

```
waiting.time <- faithful[,2]
```

The above selects the second column in the data frame `faithful` and stores the numbers in the object `waiting.time`. A histogram can be made by passing the command

```
hist(waiting.time)
```

The histogram below will then appear



11 Importing foreign data

Suppose your data is stored in an SPSS file named `foo.sav` in the root of drive X. How to get these data in R? In R there are utilities to bring this about. However, these utilities are not loaded by default when you start the program. You must first load the library `foreign` by running the command:

```
library(foreign)
```

Once the library is loaded you have for example the function `read.spss()` at your disposal. Via this function the data stored in `foo.sav` could be loaded, say, into an object `john`:

```
john <- read.spss(file="X:/foo.sav",to.data.frame=TRUE)
```

Similarly there are the functions `read.sas()`, `read.ssd()` etcetera.

Type `library(help=foreign)` to learn about the utilities provided by this library.

12 Importing Excel data

Suppose your data is stored in an Excel file named `foo.xls` in the root of drive C. That is to say, the path to the file is `C:/foo.xls`. How to get these data in R? In R there are utilities to bring this about. However, these utilities are not loaded by default when you start the program. You first have to load the library `RODBC`. Before loading this library, however, make sure that it is installed on your machine. If not, then first install it! Then load the library:

```
library(RODBC)
```

Now suppose that you want to store the first three sheets of the Excel file `foo.xls` in three objects `john`, `peter` and `nicholas`. This can be done as follows:

```
chan <- odbcConnectExcel("C:/foo.xls")
john <- sqlFetch(chan,"Sheet1")
peter <- sqlFetch(chan,"Sheet2")
nicholas <- sqlFetch(chan,"Sheet3")
close(chan)
```

Don't forget to close the channel! That is to say, don't forget to finish the operation by the command

```
close(chan)
```

If you forget this, the file `foo.xls` could get damaged.

The library `RODBC` provides more utilities then just the import of Excel files. Type

```
library(help=RODBC)
```

to get an impression of the tools in this library.

13 Reading in tabulated data

How to read in tabulated data to R? For example, how can one put in data to R that is given in the form of a frequency table like the one below?

score	frequency
A	11
AB	8
B	7
O	9

One could proceed as follows: First define an object `bloodtypes`

```
bloodtypes <- c("A","AB","B","O")
```

Mind the quotation marks! Then define a frequency vector like this

```
frq <- c(11,8,7,9)
```

Then pass the following command

```
mydata <- rep(bloodtypes,frq)
```

To see the result of your actions, type

```
mydata
```

and strike the key `Enter`. Next, try to get your frequency table back by passing the command

```
table(mydata)
```

14 Generating random data

Suppose you want to do some exercises with normally distributed data, but you don't have such data at hand. Then R can generate some (artificial) data for you. Just to have something to work with. As an example, by passing the command

```
x <- rnorm(100,170,5)
```

a sequence of 100 numbers is stored into an object `x`. The sequence will show a normally distributed structure. The mean and the standard deviation of the sequence will be *approximately* equal to 170 and 5 respectively.
How is such a sequence generated? This process could be thought to be something like this: Somewhere in R there is stored a very very long sequence of numbers, a sequence that is perfectly normally distributed with mean 170 and standard deviation 5. By the command passed above, R is drawing at random 100 numbers from this sequence. See the section Probability distributions for more info about the function `rnorm()` and its relatives.

15 Mean and median

In R it is very easy to obtain the mean or the median of a sequence of numbers. As an example, type

```
x <- c(4,2,5,1,4,8,1)
```

By this the numbers $4, 2, 5, 1, 4, 8, 1$ are stored in the object `x`. To get their mean, pass the command

```
mean(x)
```

To get a trimmed mean, type something like

```
mean(x, trim=0.025)
```

A total amount of 5% is then trimmed from your data and the mean is taken over the remaining part. If you want to see the median of the sequence $4, 2, 5, 1, 4, 8, 1$, type

```
median(x)
```

16 Variance

In R it is very easy to obtain the variance or the standard deviation of a sequence of numbers. As an example, type

```
x <- c(4,2,5,1,4,8,1)
```

By this the numbers $4, 2, 5, 1, 4, 8, 1$ are stored in the object `x`. To get their variance, pass the command

```
var(x)
```

If you want to see the standard deviation of $4, 2, 5, 1, 4, 8, 1$, type

```
sd(x)
```

Alternatively you could take the square root of the variance by passing the command `sqrt(var(x))`.

17 Transforming data

Given some sequence of numbers in R, how to transform it? As an example, define a sequence `x` by

```
x <- c(4,2,5,1,4,8,1)
```

How can one determine a sequence `y` that contains the squares of the numbers $4, 2, 5, 1, 4, 8, 1$? This can be done as follows:

```
y <- x^2
```

Similarly one can determine a sequence `z`, containing the exponentials of the numbers in `x`, like this

```
z <- exp(x)
```

In the same way one can transform the sequence `x` for example by means of the functions `log()`, `sin()`, `cos()` and so on.

18 Quantiles and boxplots

In R one can quickly obtain obtain quantiles of a sequence of numbers. To illustrate this, first generate some data and store it into the object `x` :

```
x <- rnorm(30,175,10)
```

By this a sequence of 30 numbers is stored in the object `x`. To get the quartiles of the sequence, pass the command

```
quantile(x)
```

If you want to see the associated boxplot, type

```
boxplot(x)
```

To get the quantile belonging to the fraction 0.35 (or the 35^{th} percentile), type

```
quantile(x,probs=0.35)
```

19 Making your own functions

In R virtually all elementary mathematical functions are available by default. Nevertheless you may very well encounter situations where you need a user-defined function. Suppose, for example, that you want to have a function

$$x \mapsto \frac{8}{1+x^2}\, e^{-2x}$$

available in R. To make such a function in your workspace, first make a decision how to name the function. If you decide to name it, say, `john` then pass the command

```
john <- function(x) {(8/(1+x^2))*exp(-2*x)}
```

By this action the function `john` is available in your workspace. As an example, to get the value of `john` for `x` = 1.2, pass the command

```
john(1.2)
```

The value of the function `john` in $x = 1.2$ is then printed to the screen. The function `john` allows for vectorial input. As an example, define a vector `myx` like this

```
myx <- c(1.2,1.7,2.2,2.5,2.9)
```

Now pass the command

```
john(myx)
```

This will print a set of function values to the screen. These function values, of course, can also be stored in, say, an object `y` :

```
y <- john(myx)
```

A plot of the function values could then be obtained like this:

```
plot(myx,y)
```

In the section Plotting curves it is explained how you can make nice graphs of arbitrary functions.

20 Saving functions

Functions can be saved for usage in subsequent sessions. Saving it in the root of drive X can be done like this

```
save(file="X:/myfun.RData",john)
```

By this action the function `john` is saved in a file named `myfun.RData`. You can save more than just one function in a file. If there were two more functions in your workspace, say `peter` and `nicholas`, then the three functions could be saved in one file as follows

```
save(file="X:/myfun.RData",john,peter,nicholas)
```

By this the two functions are saved in a file named `myfun.RData` in the root of drive `X`. In a subsequent session you can load the two functions in your workspace like this

```
load("X:/myfun.RData")
```

The functions `john`,`peter`and `nicholas` are then again at your disposal.

The code (or syntax) that defines functions can also be stored in plain text files. For example, by means of a text editor one could save the code

```
john <- function(x) {(8/(1+x^2))*exp(-2*x)}
peter <- function(x) {0.5+log(x/(1+x))}
nicholas <- function(x) {1+sin(3*x)}
```

in a text file named `myfun.txt`. On a Windows machine this can be done, for example, in the program 'WordPad'. It can also be done in the program 'MS-Word' but then you must save the result as a plain text file, *not* as a file in MS document format! To read in your function from the text file `myfun.txt` to your workspace, pass the command

```
source(file="X:/path/to/myfun.txt")
```

Here `X` is the drive on which the file `myfun.txt` resides, followed by the complete path to this file.

21 Plotting curves

Suppose you have produced some graphics with R and then you decide to add some curve to it. In R this can be done by means of the plot utility `curve()`. As a first example, suppose that you want to plot the graph of the function $x \mapsto \exp(-x)$ over the interval $[0, 6]$. This can be done as follows:

```
curve(exp(-x),xlim=c(0,6))
```

Next, you can add the graph of the sinus function to the plot created above like this

```
curve(sin(x),xlim=c(0,6),add=TRUE)
```

Normal curves can be plotted by means of the function `dnorm()`. For example

```
curve(dnorm(x,160,10),xlim=c(130,180))
```

plots a normal curve with mean $\mu = 160$ and standard deviation $\sigma = 10$. The plot utility `curve()` can also be used to add normal curves to a histogram. To illustrate this, first generate some data to work with:

```
height <- rnorm(100,175,5)
```

Next, make a histogram of the object `height` by running the command

```
hist(height,freq=FALSE)
```

The option `freq=FALSE` is used here to indicate that, rather than setting out the frequencies on the vertical axis, this axis must be scaled such as to result in a histogram with total area equal to 1. A normal curve (of maximal fit to the histogram) can be added as follows. First store the mean and the standard deviation of `height` in two objects, for example in `m` and `s`, like this

```
m <- mean(height)
s <- sd(height)
```

Then pass the command

```
curve(dnorm(x,m,s),add=TRUE)
```

Of course you can also plot the functions you created yourself. If you created for example the functions `john`, `peter` and `nicholas`, as described in the section Making functions, then you could plot them as curves in one figure like this

```
curve(john(x),xlim=c(1,5))
curve(peter(x),xlim=c(1,5),add=TRUE)
curve(nicholas(x),xlim=c(1,5),add=TRUE)
```

As another example, a normal density curve could also be plotted by first making a user-defined function `nicholas` :

```
nicholas <- function(x) {dnorm(x,160,10)}
```

To plot it, pass the command

```
curve(nicholas(x),xlim=c(130,180))
```

Just try it out to feel how it works!

22 Plotting straight lines

Suppose you have produced some graphics with R and then there is suddenly a wish to add some straight line to your graph. This can be done via the function `abline()`. As an example to show how it works, first pass the commands

```
u <- c(1,2,3,4)
v <- u^2
plot(u,v)
```

This will create a scatter plot with 4 points. Adding to this plot the straight line with equation

$y = 1 + 2x$

can be done like this

```
abline(1,2)
```

After striking the key `Enter` the line will be there! To plot a vertical line, at level $\mathtt{x} = 2.5$, do the following:

```
abline(v=2.5)
```

Similarly, a horizontal line at level $\mathtt{y} = 12$ can be added like this

```
abline(h=12)
```

Note that the above could also be brought about by passing the command `abline(12,0)`.

23 Kolmogorov-Smirnov plots

How to make a KS-plot in R? To illustrate this, consider the flood size data of the Ocmulgee River and store it into the (vectorial) object `flood` like this

```
flood <- c(50,12,16,20,17,13,61,26,33,38)
```

To make the empirical cumulative frequency function (ecdf) that belongs to the data, run the command

```
john <- ecdf(flood)
```

To get a plot, do this

```
plot(john,verticals=TRUE)
```

If you want to include labels on the x-axis, then do something like

```
plot(john,verticals=TRUE,xlab="Size of Flood")
```

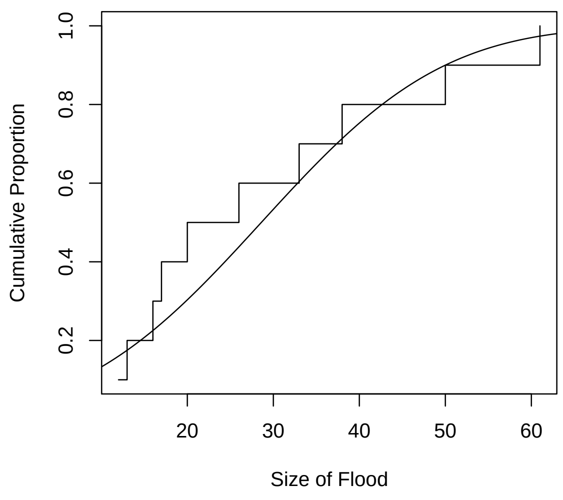
Similarly a label can be attached to the y-axis.
To add the associated normal cdf to the plot created above, procede as follows: First store the mean and the standard deviation of `flood` in two objects, for example in `m` and `s`, like this

```
m <- mean(flood)
s <- sd(flood)
```

Then pass the command

```
curve(pnorm(x,m,s),add=TRUE)
```

The figure below will then be created



24 Kolmogorov-Smirnov distance

The KS-distance is a component in the so-called Kolmogorov-Smirnov test, which will be discussed in more detail in a later section. To show how the KS-distance to normality can be obtained, store (to have some data) the flood size data of the Ocmulgee River into the object `flood` as follows:

```
flood <- c(50,12,16,20,17,13,61,26,33,38)
```

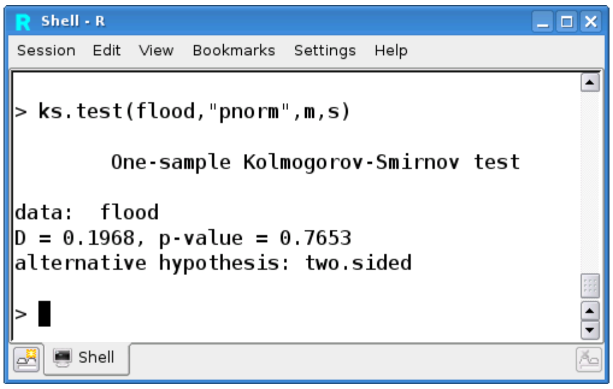
Then store the mean and the standard deviation of `flood` in two objects, for example in `m` and `s`, like this

```
m <- mean(flood)
s <- sd(flood)
```

Finally, run the command

```
ks.test(flood,"pnorm",m,s)
```

The following output will then be returned



```
> ks.test(flood,"pnorm",m,s)

        One-sample Kolmogorov-Smirnov test

data:  flood
D = 0.1968, p-value = 0.7653
alternative hypothesis: two.sided

> 
```

The KS-distance is denoted by the character `D` and can be read off to have the value 0.1968. In the output you can also find the p-value.

25 QQ-plots

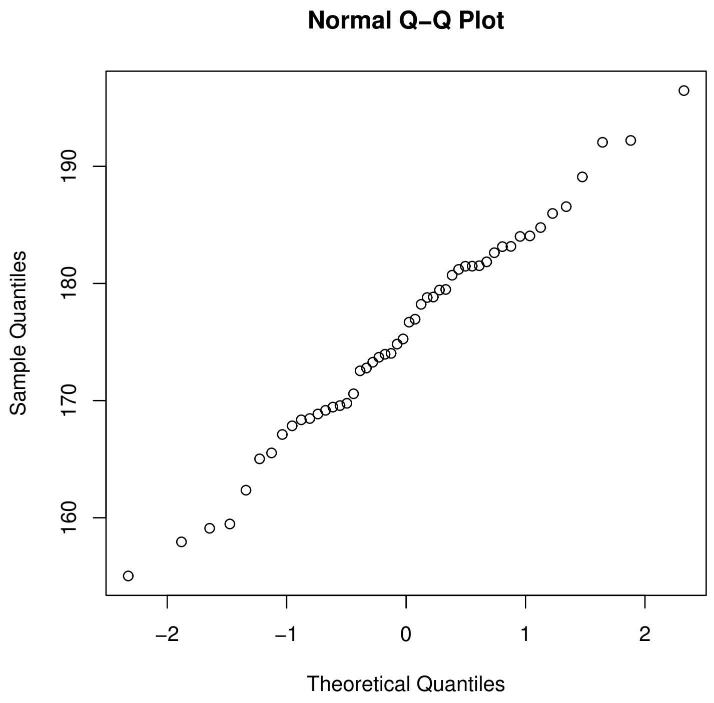
QQ-plots can easily be made in R. As an example, generate some data to work with:

```
x <- rnorm(50,175,10)
```

Then pass the command

```
qqnorm(x)
```

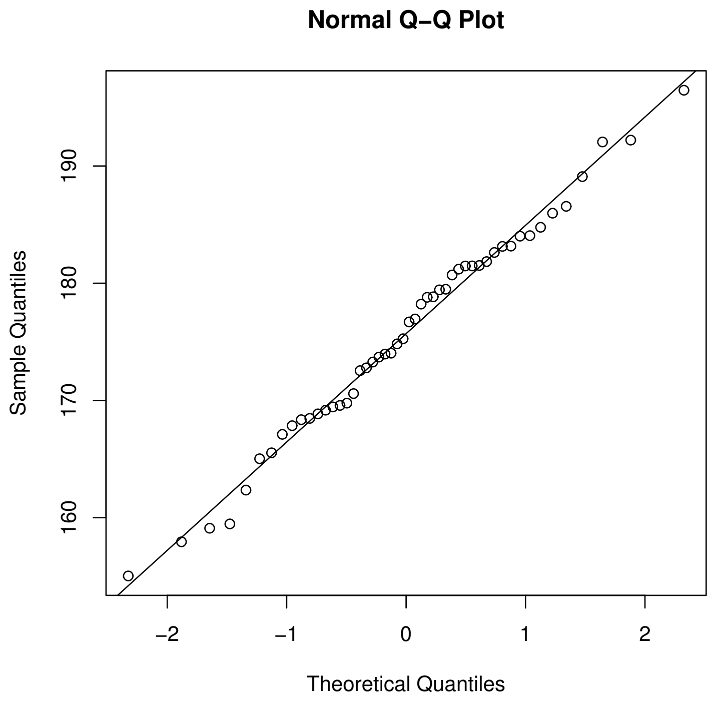
The result will be a graph like the one below:



A reference line in your QQ-plot can be added by passing the command (presumed your variable is named `x`)

```
abline(mean(x),sd(x))
```

The QQ-plot will then look like this



26 Probability distributions

In R you can deal with (for example) the t-distributions via the functions

```
dt()
pt()
qt()
rt()
```

The function `dt()` returns values of t-densities. You could use this function, for example, to make a graph of a t-density:

```
curve(dt(x,8),xlim=c(-3,3))
```

This would create a graph of the t-density with 8 degrees of freedom.

The function `pt()` returns the cumulative probabilities that emanate from t-distributions. For example, the command

```
pt(1.32,9)
```

would return the size of a left probability tail with cut-off point 1.32 in a t-distribution with 9 degrees of freedom.

The function `qt()` does the converse of the function `pt()`. So the command

```
qt(0.95,12)
```

returns the 95th percentile of a t-distribution with 12 degrees of freedom.

Finally, the function `rt()` generates samples from t-distributed populations. For example, the command

```
rt(100,12)
```

would return a sample of length 100 from a t-distributed population with 12 degrees of freedom.

A scheme of probability distributions and their abbreviations in R is given below:

Distribution	Abbreviation
Normal	`norm`
Student's T	`t`
Binomial	`binom`
Poisson	`pois`
Gamma	`gamma`
Beta	`beta`
Uniform	`unif`
Exponential	`exp`
Geometrical	`geom`
Fisher	`f`
χ^2	`chisq`

These distributions can all be used in combination with the prefixes `d`, `p`, `q` and `r`.

Prefix	Function
`d`	returns values of the density
`p`	returns cumulative probabilities
`q`	returns quantiles
`r`	returns random samples

For example, for normal distributions one has the functions `dnorm()`, `pnorm()`, `qnorm()` and `rnorm()`.

27 Plotting probability densities

Suppose you want a plot of the χ^2 - density with 3 degrees of freedom over an `x`-interval that runs from 0 to 18. Then just pass the command

```
curve(dchisq(x,3),xlim=c(0,18))
```

If you want to add the χ^2 - density with 5 degrees of freedom to the plot, then run

```
curve(dchisq(x,5),add=TRUE)
```

Perhaps you want have the graph of the χ^2 - density with 10 degrees of freedom also added to the plot. Then just run

```
curve(dchisq(x,10),add=TRUE)
```

In this way χ^2 - densities can also be added to histogram plots.

If you want to plot the cumulative distribution function of a χ^2 - distribution, then do the same as sketched above, thereby replacing the function `dchisq()` by `pchisq()`.

28 Confidence intervals for a population mean

Confidence intervals can be obtained by applying a t-test. To illustrate one thing and another, make some artificial data. For example, some artificial weight data

```
weight <- c(3.7, 7.4, 8.6, 5.3, 4.4)
```

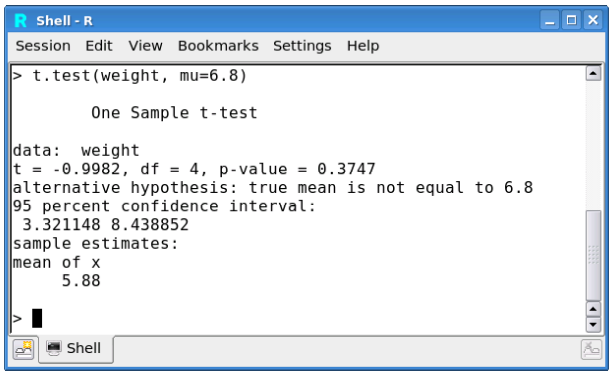
To carry out a t-test with null hypothesis

$H_0 : \mu = 6.8$

pass the command

```
t.test(weight, mu=6.8)
```

Then strike the key `Enter`. The output will then immediately appear on the screen. It is of the form depicted below:



```
> t.test(weight, mu=6.8)

        One Sample t-test

data:  weight
t = -0.9982, df = 4, p-value = 0.3747
alternative hypothesis: true mean is not equal to 6.8
95 percent confidence interval:
 3.321148 8.438852
sample estimates:
mean of x
     5.88

>
```

In the output above you can see that a 1-sample t-test was carried out (t-tests come in several flavours). As you can see, the output also contains a 95% confidence interval for the population mean. It is the interval $(3.321148; 8.438852)$. If you are only interested in a confidence interval, then it is not important which null hypothesis you choose.

29 The 1-sample t-test

In R all flavors of the t-test can be conducted by means of the command `t.test()`. To illustrate how a 1-sample t-test can be carried out, first make some artificial height data to work with

```
height <- c(174,165,179,189,172,161)
```

To test for example the null hypothesis

$H_0 : \ \mu = 175$

just pass the command

```
t.test(height,mu=175)
```

In the output that is returned (after striking the key `Enter`) you can find for example a p-value and a confidence interval for the population mean.

30 The 2-sample t-test

To illustrate how to conduct a 2-sample t-test you need two independent datasets. Make them in an artificial way and think of them as being height measurements in Holland and France:

```
holland <- c(174,165,179,189,172,161)
france <- c(173,180,172,155,169)
```

To figure out whether these two samples differ significantly in mean, just pass the command

```
t.test(holland,france)
```

Among the output there is a p-value and a confidence interval for the difference in population means in Holland and France.

If the heights would have been put in as one vector, say, a vector `height` with a grouping variable `group` to define the groups, then the t-test can simply be run as

```
t.test(height~group)
```

31 The paired-sample t-test

A paired-sample t-test can be conducted in R in very much the same way as a 2-sample t-test. To illustrate this, first make some data to work with

```
height1 <- c(174,165,179,189,172,161)
height2 <- c(175,167,179,189,173,162)
```

Think of these data as being six pairs of height measurements. For example, the height of six adolescents could be measured at two points in time. To check whether there is a significant change in height, pass the following command

```
t.test(height1,height2,paired=TRUE)
```

In the output that is returned (after striking the key `Enter`) you can find a p-value and a confidence interval for the mean difference in height.

32 Kolmogorov-Smirnov tests

To show how a KS-test on normality can be run in R, store (to have some data) the flood size data of the Ocmulgee River into the variable `flood` as follows:

```
flood <- c(50,12,16,20,17,13,61,26,33,38)
```

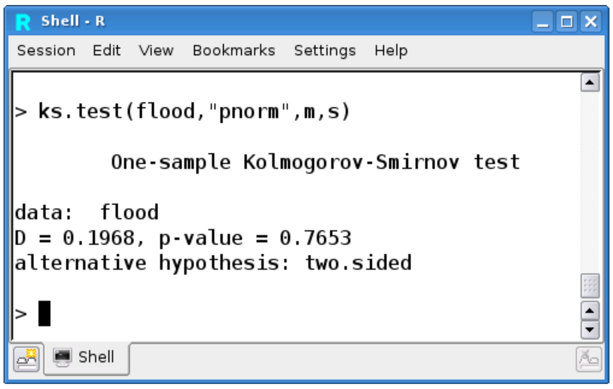
Then store the mean and the standard deviation of `flood` in two variables, for example in `m` and `s`, like this

```
m <- mean(flood)
s <- sd(flood)
```

Finally, run the command

```
ks.test(flood,"pnorm",m,s)
```

R will return some info then, including a p-value and the KS-distance.



```
> ks.test(flood,"pnorm",m,s)

        One-sample Kolmogorov-Smirnov test

data:  flood
D = 0.1968, p-value = 0.7653
alternative hypothesis: two.sided

> 
```

The KS-distance is denoted by the character `D` and can be read off to have the value 0.1968. The p-value is 0.7653.

If, for certain reasons, you would like to check whether the sample `flood` could have had its origin in a t-distributed population with 6 degrees of freedom, then pass the command

```
ks.test(flood,"pt",6)
```

R will return a KS-distance and a p-value.

Similarly, if you want to check whether `flood` is a sample from an exponentially distributed population with mean 3, run the command

```
ks.test(flood,"pexp",3)
```

R will return a KS-distance (to an exponential distribution with parameter 3) and a p-value.

If you want to check whether the distribution of `flood` differs significantly from an F_{10}^{8} -distribution, pass the following command:

```
ks.test(flood,"pf",8,10)
```

R will return a KS-distance (to an F_{10}^{8} -distribution) and a p-value.

33 Fisher's variance test

Suppose you have in your workspace two sequences of measurements as to lung capacity of 8-year old children. One sequence, say, `lungcapgirls` contains the measurements of the girls and the other, `lungcapboys`, those of the boys. After inspection it appears that both sequences pass the KS-test as to normality. Next you want run Fisher's variance test to check whether the variance in both sequences differs significantly. In R you can do this by simply passing the command

```
var.test(lungcapgirls,lungcapboys)
```

After striking the key `Enter` you will have a p-value returned.

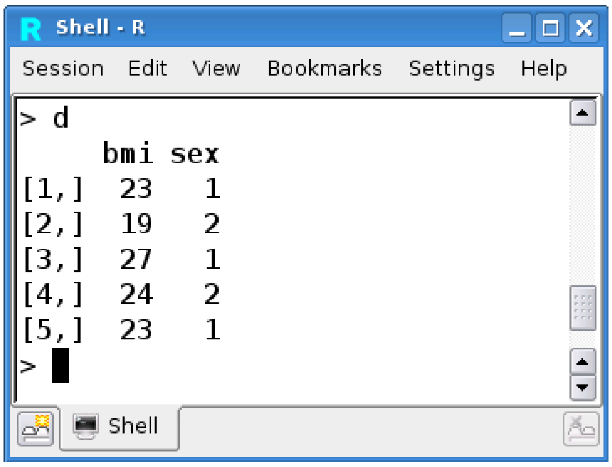
If all lung capacities of the girls and boys were stored in one column, say a column `lungcap`, and if the groups were defined by a grouping variable, say a variable named `sex`, then one could run Fisher's variance test as follows:

```
var.test(lungcap~sex)
```

Th test results will be returned after striking the key `Enter`.

34 Splitting datasets

Suppose the dataset `d` below is given:



```
> d
     bmi sex
[1,]  23   1
[2,]  19   2
[3,]  27   1
[4,]  24   2
[5,]  23   1
>
```

You want to split up the dataset into two datasets `d1` and `d2` according to the grouping variable `sex`. This can be done as follows by using the function `subset()` :

```
d1 <- subset(d,sex==1)
d2 <- subset(d,sex==2)
```

To extract those rows for which the variable `bmi` is larger than 22, run

```
d3 <- subset(d,bmi>22)
```

Another way to split the dataset `d` is by using the function `split()` in the following way:

```
attach(d)
ds <- split(d,sex)
```

By this action a so-called ‘list’ is created (see the section Numbers, strings and objects in R). The list contains two components: a component `ds$1` and a component `ds$2`. To extract the measurements with sex codes 1 and 2 as separate datasets from the list, just pass the command

```
d1=ds$1
d2=ds$2
```

Note that if the coding were `"male"` and `"female"` rather than 1 and 2, then the components of the list `ds` would have been `ds$male` and `ds$female`.

35 The 1-way ANOVA

Running ANOVA’s in R is very easy. To give an example, load some built-in data into you workspace:

```
data(PlantGrowth)
```

This dataset arose from an experiment on plant growth under several conditions. There is a wish to check whether the conditions resort significantly in effect. To bring this about, first have a look at the headers of the dataset (they are `weight` and `group`). If you want to use these headers as if they were variables in your workspace, pass the command:

```
attach(PlantGrowth)
```

Then run the following command (where PG stands for Plant Growth):

```
PG <- lm(weight~factor(group))
```

By this action R ran the ANOVA mentioned above. The results are stored in an object which you gave the name `PG`. To view the results, type

```
anova(PG)
```

A list of residuals can be extracted from the object `PG` like this

```
e <- residuals(PG)
```

By the above the residuals of the ANOVA are stored into the variable `e`. This variable `e` can then be checked on normality.

To check homogeneity of variances one could run Bartlett's test:

```
bartlett.test(weight~factor(group))
```

The null hypothesis in Bartlett's test is that there is homogeneity indeed. Bartlett's test is a generalization of Fisher's variance test for situations in which there are more than two groups. Both Bartlett's and Fisher's test lean on normality assumptions and are very sensitive to deviations in this. A more robust alternative to these tests is Levene's variance test. This test is part of the library `car`. Once this library is loaded, Levene's test could, in this example, be run by passing the command

```
levene.test(PG)
```

Equivalently one could run

```
levene.test(weight~factor(group))
```

Be aware that the p-values returned by Bartlett's and Levene's test may be very different! Also be aware that R uses a version of the Levene's test that differs from the version used in, for example, SPSS.

In R multiple comparisons can be done like this

```
pairwise.t.test(weight,factor(group))
```

This action will return a kind of a cross table from which the corrected p-values of all comparisons can be read off.

36 The 2-way ANOVA

A 2-way ANOVA in R can be run similar to a 1-way ANOVA. To give an example, load some built-in data into you workspace:

```
data(morley)
```

This is the dataset that emerged from the famous Michelson-Morley experiments, in which the speed of light was measured. There is a wish to check whether the factors `Run` and `Expt` resort effect. Interaction is excluded beforehand. To bring this about, first have a look at the headers of the dataset (they are `Expt`, `Run` and `Speed`). These headers can be revealed by using the function `head()` or `tail()`. If you want to use these headers as if they were variables in your workspace, pass the command:

```
attach(morley)
```

Then run the following command (where `MM` stands for Michelson-Morley):

```
MM <- lm(Speed~factor(Expt)+factor(Run))
```

If you want to include the possibility of interaction in your 2-way ANOVA, then run the command

```
MM <- lm(Speed~factor(Expt)+factor(Run)+factor(Expt):factor(Run))
```

Equivalently, the above could be run by passing the command

```
MM <- lm(Speed~factor(Expt)*factor(Run))
```

However, on this dataset this would not work: For a 2-way ANOVA with interaction each cell must contain at least two elements. This condition is not met here ...

Levene's test (in the library `car`), to check homogeneity of variance, could be carried out by simply passing the command

```
levene.test(MM)
```

Equivalently one could run

```
levene.test(Speed~factor(Expt)*factor(Run))
```

Again, for this particular dataset there is a problem in running the above because each cell contains only one measurement. For this reason one cannot talk about the variance of the measurements in a cell.

37 Wilcoxon's signed-rank test

First make some suitable data to work with

```
x <- c(0.80, 0.83, 1.89, 1.04, 1.45)
y <- c(1.15, 0.88, 0.90, 0.74, 1.21)
```

Assume that the data in these two sequences are paired, that is to say, 0.80 belongs to 1.15, 0.83 belongs to 0.88 and so on. Run a Wilcoxon test by passing the command

```
wilcox.test(x,y,paired=TRUE)
```

If you want exact p-values, run

```
wilcox.test(x,y,paired=TRUE,exact=TRUE)
```

38 The Mann-Whitney test

In R the Mann-Whitney test is referred to as a Wilcoxon rank-sum test. To illustrate how to run this test, first make some suitable data to work with:

```
x <- c(0.80, 0.83, 1.89, 1.04, 1.45, 1.38)
y <- c(1.15, 0.88, 0.90, 0.74, 1.21)
```

Then run the command

```
wilcox.test(x,y)
```

To give another example, load the built-in dataset `sleep` into your workspace

```
data(sleep)
```

This dataset contains two columns with headers `extra` and `group`. The column with header `group` serves as a grouping variable indeed. If you want to use the headers as if they were variables in your workspace, pass the command

```
attach(sleep)
```

Then run a Mann-Whitney test like this

```
wilcox.test(extra~group)
```

If you want exact p-values, run

```
wilcox.test(extra~group,exact=TRUE)
```

39 The Kruskal-Wallis test

First load some suitable built-in dataset:

```
data(chickwts)
```

This dataset contains two columns with headers `weight` and `feed`. The column with header `feed` could serve here as a grouping variable. If you want to use these headers as if they were variables in your workspace, pass the following command

```
attach(chickwts)
```

To run a Kruskal-Wallis test, pass the command

```
kruskal.test(weight~feed)
```

The Kruskal-Wallis test can be regarded as a nonparametric alternative to a 1-way ANOVA.

40 The runs test

The runs test is not contained in R-base. You will have to load the library `tseries`. Once having done this, it is possible to do a runs test in R. Given a sequence of symbols `x` one can do it like this:

```
runs.test(x)
```

If R starts complaining that the input is not a factor, then pass the command:

```
runs.test(factor(x))
```

R will return some info, among which a p-value.

41 Simple linear regression

To learn how to run a simple linear regression in R, put in the following dataset:

```
x <- c(1,2,3,4,5)
y <- c(3,3,5,7,7)
```

Then pass the command

```
lm(y~x)
```

The regression coefficients will be returened by this action. Wiser than the above would be to run a command of type

```
myregression <- lm(y~x)
```

By this action all results of the regression analysis (the residuals included) are stored into an object that you have given the name `myregression`. A summary of the whole regression analysis can be obtained like this

```
summary(myregression)
```

Confidence intervals for the coefficients can be extracted from the object `myregression` by passing the command

```
confint(myregression)
```

To extract from the object `myregression` the residuals, pass the command

```
e <- residuals(myregression)
```

By the above the residuals are stored into an object `e` and can then be used for further analysis.

To get the fitted values stored in an object `yhat` do the following

```
yhat <- fitted(myregression)
```

To get a prediction for the value of y when it is given that $x = 3.5$, first store the value of 3.5 in the object `x0` like this:

```
x0 <- data.frame(x=3.5)
```

Then run the command

```
predict(myregression,newdata=x0,interval="pred")
```

This will provide you with the desired prediction intervals.
To illustrate how you can run a regression through the origin, type in some data to work with

```
x <- c(1,2,3,4,5)
y <- c(3,3,5,7,7)
```

Recall that an ordinary simple regression can be run by passing the command

```
myregression <- lm(y~x)
```

All results of the regression analysis are then stored into an object that you have named `myregression`.
To run a simple linear regression through the origin, pass the command

```
myregression <- lm(y~x-1)
```

All information can then be extracted from the object `myregression` as was sketched above.

42 Multiple linear regression

In the previous section it was explained how to run a simple linear regression in R. If you want to explain your variable `y` by two `x`-variables in your regression model, say, `x1` and `x2`, then pass the command

```
myregression <- lm(y~x1+x2)
```

If you don't want the constant in your model, this can be brought about as follows:

```
myregression <- lm(y~x1+x2-1)
```

You are then running a multiple linear regression through the origin. If an `x`-variable, say `x2`, happens to categorical, then run

```
myregression <- lm(y~x1+factor(x2))
```

43 Regression plots

In the following it is assumed that a simple regression analysis was already run:

```
myregression <- lm(y~x)
```

A basic regression plot consists of a scatter plot with the regression line in it. This can be obtained like this:

```
plot(x,y)
abline(myregression)
```

A residual plot can be obtained as follows:

```
e <- residuals(myregression)
yhat <- fitted(myregression)
plot(yhat,e)
```

```
plot(myregression)
```

The function `segments()` plots adds line segments to scatter plots. This function can also be of use in regression plots. To discover how it works, define

```
u1 <- 1
u2 <- 4
v1 <- 5
v2 <- 1
```

Then pass the command

```
plot(c(u1,u2),c(v1,v2))
segments(u1,v1,u2,v2)
```

You will see that the function `segments()` adds line segments to a scatter plot.

A scatter plot in which the regression line and the residuals are visible can be obtained like this

```
plot(x,y)
segments(x,y,x,yhat)
```

Type `?segments` to get some more info about the function `segments()`.

44 Generating random regression data

The three conditions in linear regression are:

- In the population y must be quasi linear in x.
- In the population the residuals must be normally distributed.
- There must be homoscedasticity.

How to generate random regression datasets that satisfy the above conditions? For example, how can one generate samples from a population with regression line $y = 2 + 3\,x$? This can be done as follows. First create a sequence of x -values, for example like this:

```
x <- 1:25
```

Then create the corresponding y -values:

```
y <- 2 + 3*x + rnorm(25,0,2)
```

Now the variables `x` and `y` define 25 points in the plane. These 25 points can be thought to be a sample from a population that meets the three regression conditions. The population has a regression line $y = 2 + 3\,x$ and the residuals relative to this line are normally distributed with standard deviation 2.

Similarly one can create such datasets in multiple linear regression. As an example, one could create

```
x1 <- 1:24
x2 <- rep(1:12,2)
y <- 2 + 3*x1 - x2 + rnorm(24,0,3)
```

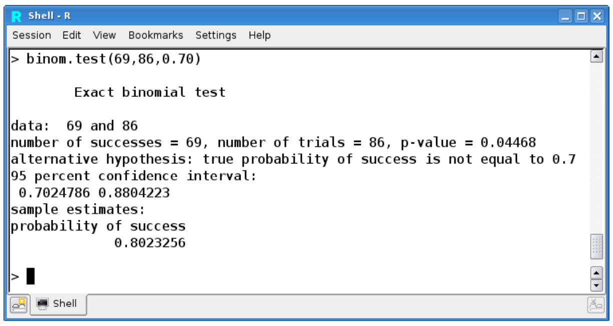
The variables `x1`, `x2`, `y` define 24 point in 3-dimensional space. These 24 points can be thought to be a sample from a population that meets the three regression conditions. The population has a regression equation $y = 2 + 3\,x_1 - x_2$ and the residuals relative to this equation are normally distributed with standard deviation 3.

45 Binomial tests

A tool to run a 1-sample proportion test in R is the binomial test. To illustrate how to use it, suppose that in an experiment you have drawn a sample of 86 patients from a certain population. Among these 86 patients there are 69 smokers. You want to test the null hypothesis that the population proportion of smokers be 70%. This can be done like this

```
binom.test(69,86,0.70)
```

The output below will then be returned



```
> binom.test(69,86,0.70)

        Exact binomial test

data:  69 and 86
number of successes = 69, number of trials = 86, p-value = 0.04468
alternative hypothesis: true probability of success is not equal to 0.7
95 percent confidence interval:
 0.7024786 0.8804223
sample estimates:
probability of success
             0.8023256

>
```

From the output it can be read off that the the p-value is 0.04468.

46 Proportion tests

A versatile and userfriendly tool to analyze proportions in R is the proportion test. To illustrate how to run a 1-sample proportion test, suppose that in an experiment you have drawn a sample of 86 patients from a certain population. Among these 86 patients there are 69 smokers. You want to test the null hypothesis that the population proportion of smokers be 70%. This can be done as follows:

```
prop.test(69,86,0.70)
```

Note that the above could also be settled with the binomial test. Actually, the advantage of the binomial test is that it returns exact p-values. The proportion test returns approximate p-values. The advantage of the proportion test, however, is that it is more versatile. It can, for example, be used to compare two (or even more than two) proportions. To show how the proportion test can be used to compare two empirical proportions, suppose that you have two populations of patients, say population A and population B. From population A you draw a sample of 86 patients and a count reveals that there are 69 smokers among them. Similarly, a sample of 93 patients is drawn from population B and it turns out that there are 85 smokers among them. A test to check whether the two proportions 69:86 and 85:93 differ significantly can be run like this

```
smokers  <- c(69,85)
patients <- c(86,93)
prop.test(smokers, patients)
```

To compare three smoker proportions, for example 69:86, 85:93 and 121:136, do the following

```
smokers  <- c(69,85,121)
patients <- c(86,93,136)
prop.test(smokers, patients)
```

47 Contingency tables

How to make contingency tables (matrices) in R? Just try out the following

```
x <- c(794,86,150,570)
y <- matrix(x,ncol=2)
```

The rows in matrix `y` can be assigned names in the following way:

```
rownames(y) <- c("first row","second row")
```

Similarly the columns can be assigned names

```
colnames(y) <- c("first column","second column")
```

Print the variable `y` to the screen to view the effect of the actions described above.

48 McNemar's test

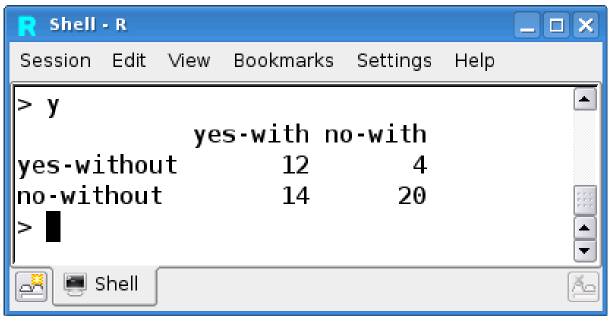
The McNemar test is easy to run in R. To have some data, put in a table like this:

```
x <- c(12,14,4,20)
y <- matrix(x,ncol=2)
```

Edit the contingency table for example like this, where ‘yes’ stands for ‘yes, the person sees relief’ and ‘with’ for ‘with 3D-glasses’:

```
colnames(y) <- c("yes-with","no-with")
rownames(y) <- c("yes-without","no-without")
```

Your contingency table is now ready for use. When you print the matrix `y` to the screen you will see something like this:

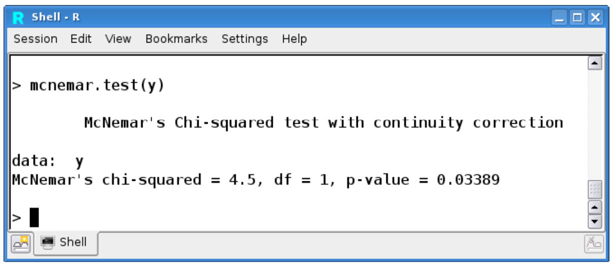


```
> y
            yes-with no-with
yes-without       12       4
no-without        14      20
>
```

McNemar’s test can now be run in this way:

```
mcnemar.test(y)
```

On the screen there will then be something returned like this



```
> mcnemar.test(y)

        McNemar's Chi-squared test with continuity correction

data:  y
McNemar's chi-squared = 4.5, df = 1, p-value = 0.03389

>
```

49 The chi-square test on goodness of fit

To learn how to run a χ^2 - test on goodness of fit, use the data below as an example:

	A	B	AB	O
Observed	56	29	58	57
Expected	68	30	46	56

Store the observed frequencies in a variable which you give for example the name `observed`:

```
observed <- c(56,29,58,57)
```

Then convert the expected frequencies into a so-called *probability vector* named (for example) `expected`, like this

```
expected <- c(68,30,46,56)
expected <- expected/sum(expected)
```

Print the vector `expected` to the screen to see the result. To run the χ^2 - test, pass the command

```
chisq.test(observed,p=expected)
```

50 The chi-square test on independence

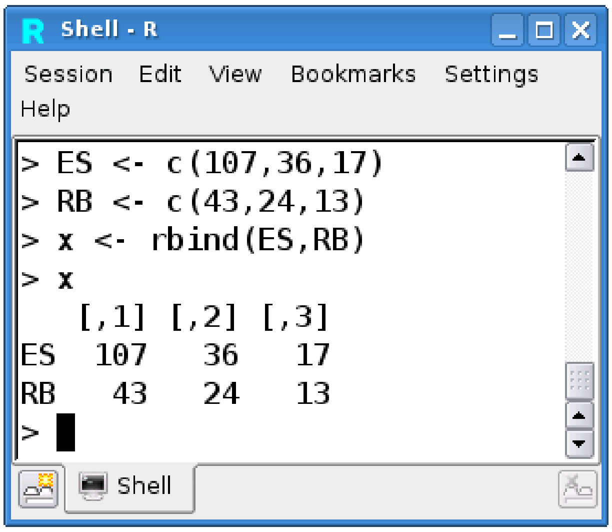
How to run a χ^2 - test in R on the next contingency table?

	A	B	AB
European Shorthair	107	36	17
Russian Blue	43	24	13

First you have to put in the table. This can be done, for example, like this

```
ES <- c(107,36,17)
RB <- c(43,24,13)
x <- rbind(ES,RB)
```

When printing x to the screen you will see something like this



```
> ES <- c(107,36,17)
> RB <- c(43,24,13)
> x <- rbind(ES,RB)
> x
   [,1] [,2] [,3]
ES  107   36   17
RB   43   24   13
> 
```

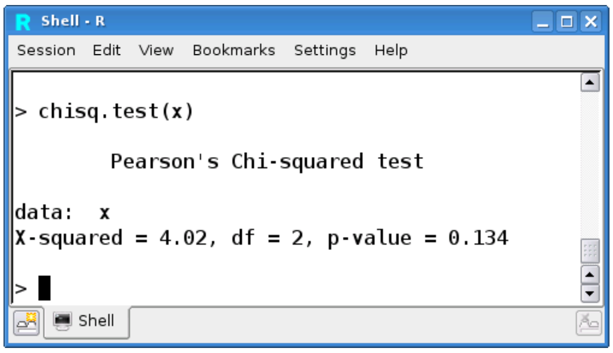
If you want (this is optional) you can assign headers to the columns, for example like this

```
colnames(x) <- c("A","B","AB")
```

Print x to the screen to view the result of this action. To run a χ^2 - test on the newborn contingency table x, simply pass the command

```
chisq.test(x)
```

The output will be something like this



```
> chisq.test(x)

        Pearson's Chi-squared test

data:  x
X-squared = 4.02, df = 2, p-value = 0.134

> 
```

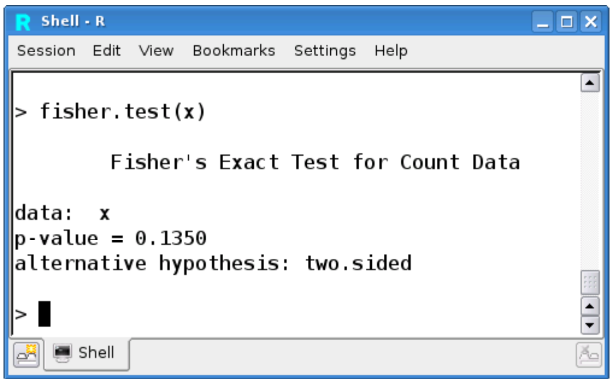
The p-value can be read off to be 0.134.

The null hypothesis is therefore maintained, that is to say, the dataset provides no evidence that there is a relation between blood type and the two cat races.

In a very similar way one could run Fisher's exact test on the contingency table `x`:

```
fisher.test(x)
```

The output below will then be returned



```
> fisher.test(x)

        Fisher's Exact Test for Count Data

data:  x
p-value = 0.1350
alternative hypothesis: two.sided

>
```

Note that the p-values returned by Fisher's exact test are exact indeed, whereas those returned by the χ^2 - test only approximate.

51 Fitting nonlinear models

To illustrate things one could start from the dataset below:

x	y
1	2
2	3
3	5
4	5
5	9

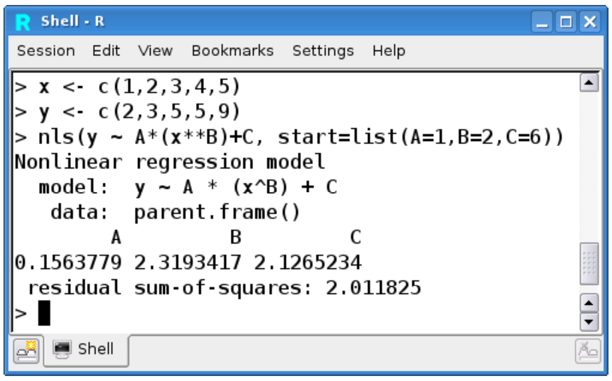
Start the program R and put in the above dataset (from console) as follows:

```
x <- c(1,2,3,4,5)
y <- c(2,3,5,5,9)
```

Then strike the key `Enter`. Next, pass the command

```
nls(y~A*(x**B)+C,start=list(A=1,B=2,C=-6))
```

After striking the key `Enter` you will see the output appear like this:



```
> x <- c(1,2,3,4,5)
> y <- c(2,3,5,5,9)
> nls(y ~ A*(x**B)+C, start=list(A=1,B=2,C=6))
Nonlinear regression model
  model:  y ~ A * (x^B) + C
   data:  parent.frame()
        A         B         C
0.1563779 2.3193417 2.1265234
 residual sum-of-squares: 2.011825
> 
```

The values for A, B and C can be read off to be 0.1564, 2.3193, 2.1265 respectively and the minimum value for SSE (presented here as the `residual sum-of-squares`) is evaluated as 2.0118.

More info about the routine `nls()` in R can be obtained by just typing

```
?nls
```

and then strike the key `Enter`. A help screen will then appear. Under Unix systems, type `q` to leave this help screen and continue in R.

52 Residual analysis

To carry out a residual analysis in R, first get your variables (say `x` and `y`) in the work space. There are many ways to do this. Just choose one. Now suppose that you want to fit the model

$$y = A * e^{-B*x}$$

Then choose a set of initial values, for example $A = 1$, $B = 2$ and pass the command

```
r <- nls(y~A*exp(-B*x),start=list(A=1,B=2))
```

After striking the key `Enter`, an object `r` is created in the work space of R. This object contains all results of the model fit, for example also the residuals. To extract the residuals from the object `r`, pass the command

```
res <- residuals(r)
```

Now the residuals are stored in the variable `res`. To see the content of the variable `res` on the screen, simply type

```
res
```

and strike the key `Enter`. There are your residuals!

Checking the residuals on normality can be done via the Shapiro-Wilk test like this

```
shapiro.test(res)
```

Homoscedasticity can be checked graphically by creating a residual scatter plot:

```
plot(x,res)
```

53 Logistic regression

Consider the dataset

height	problem
171	1
161	0
183	1
168	0
172	1
189	0
162	0
179	1

Put the dataset in like this

```
height <- c(171,161,183,168,172,189,162)
problem <- c(1,0,1,0,1,0,0)
```

In R a logistic regression can be run (in this particular case) by passing the command

```
name <- glm(problem~height,family=binomial)
```

A summary of the results of the analysis can be obtained like this

```
summary(name)
```

If, besides the variable `height`, the dataset also contained a variable `BMI`, then one could include this variable in the model like this:

```
name <- glm(problem~height+BMI,family=binomial)
```

If there were also a categorical variable `blood` in the dataset, put it into the model like this

```
name <- glm(problem~height+BMI+factor(blood),family=binomial)
```

Model reduction can be applied by running

```
step(name)
```

To get a prediction for a person with height 165 cm, a BMI of 24 and bloodtype O, create for example a variable `preddata` like this

```
preddata <- data.frame(height=165,BMI=24,bloodtype="O")
```

Then pass the command

```
predict(name,newdata=preddata,type="response")
```

If in the command above `newdata` is not specified, then R will return the probabilities relative to the cases in the dataset to which the fit was carried out.

If you want to extract the value of the likelihood of the fitted model from the object `name`, then run the command

```
logLik(name)
```

The natural logarithm of the likelihood value will then be returned.

54 Nonparametric survival analysis

Consider the following survival data

years	event
71	1
61	2
83	1
68	3
72	1
89	2
62	3
79	1

The events are classified in three categories. If the event of interest is coded by 3 then a corresponding survival object, say `mary`, can be created like this

```
mary <- survfit(Surv(years,event==3))
```

The events coded by 1 or 2 are now regarded as being censored. A Kaplan-Meier plot can be extracted from the object `mary` as follows:

```
plot(mary)
```

If you don't want the confidence bands in the plot, then remake the object `mary`, thereby using the option `conf.type = "none"`.

Next, suppose that there is a grouping variable in the dataset:

years	event	group
71	1	1
61	2	1
83	1	1
68	3	1
72	1	1
89	2	2
62	3	2
79	1	2

If you want to compare the Kaplan-Meier curves of the groups in a logrank test, then make an object, say `jane`, as follows

```
jane <- survdiff(Surv(years,event)~group)
```

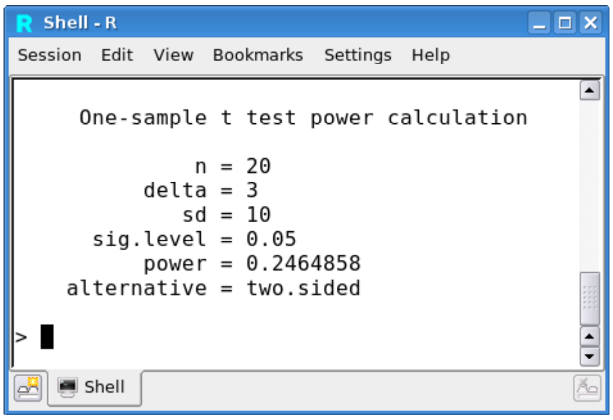
If you don't specify the event of interest, then by default the event coded by 1 is taken. By simply typing `jane` you will see the results of the logrank test on your screen.

55 Power and sample size (I)

In R there there are some extremely useful functions as to determining power, sample sizes etcetera. One of them is designed for t-tests. It is the function `power.t.test()`. To illustrate how to use this function, suppose you want to test a (null) hypothesized population mean $\mu = 120$. Your sample size is 20 and you want to test on a significance level of $\alpha = 0.05$. It is known that the standard deviation of the population is 10. You want to know the power of the test against the alternative $\mu = 123$. To get this info, run

```
power.t.test(n=20,delta=3,sd=10,sig.level=0.05,type="one.sample")
```

In the arguments of `power.t.test()`, `n` is the sample size, `delta` is the difference between the null hypothesized and the alternative mean, `sd` is the standard deviation of the population and `type` is the type of the t-test. You will then get the next output:



```
     One-sample t test power calculation

              n = 20
          delta = 3
             sd = 10
      sig.level = 0.05
          power = 0.2464858
    alternative = two.sided

> 
```

From this output you can read off that the power in the alternative $\mu = 123$ is 0.2465.

The function `power.t.test()` can also be used to compute sample sizes necessary to get things significant. Suppose, for example, that the null hypothesized population mean is $\mu = 120$. You expect in your sample a mean of about 125, a standard deviation of about 7 and you want to have things significant on the 5% level and you want a power of 0.8 or more in the alternative $\mu = 125$. What sample size must be chosen to have the sample mean significantly different from the null hypothesized mean? Run the next command to find it out:

```
power.t.test(delta=5,sd=7,sig.level=0.05,power=0.8,
                                              type="one.sample")
```

In the output that is returned after striking the key `Enter` you can find a minimum value for the sample size in question.

Similar to the function `power.t.test()` there are the functions `power.anova.test()`, `power.prop.test()`. They can be used when dealing with power and sample size problems in ANOVA and proportion tests.

56 Power and sample size (II)

In the previous section it was described how to use the three functions `power.t.test()`, `power.prop.test()` and `power.anova.test()`. More extensive tools to compute power and sample sizes are provided by the library `pwr`. This library is not contained in R-base; it must be installed before using it. To install it, first make sure that you are connected to the internet. Then run R with Administrator privileges and pass the command

```
install.packages("pwr")
```

After a successful installation the library can be loaded into R by running the command

```
library(pwr)
```

The contents of the library can be viewed by passing the command

```
help(pwr)
```

Among the functions in the library `pwr` there is the function `pwr.t.test()`. This function can be used in very much the same way as the function `power.t.test()`. As an example, one could run

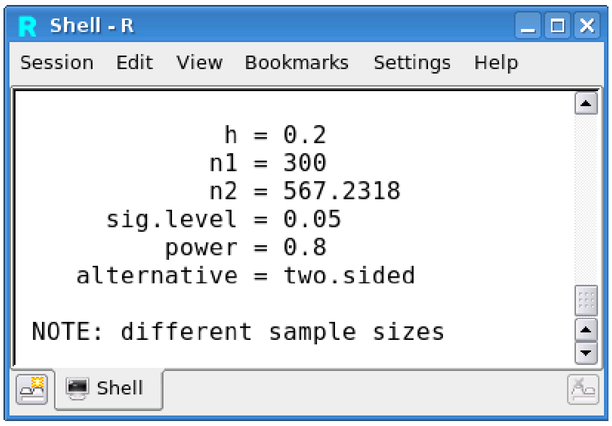
```
pwr.t.test(n=20,d=3,sig.level=0.05,type="one.sample")
```

In the arguments of `pwr.t.test()`, `n` is the sample size, `d` is the effect size (the difference between the null hypothesized and the alternative mean divided by the standard deviation) and `type` is the type of the t-test.

Another tool in this library is the function `pwr.2p2n.test()`. Use this function when dealing with 2-sample proportion tests with possible unequal sample sizes. A call to this function could be as follows:

```
pwr.2p2n.test(h=0.2,n1=300,sig.level=0.05,power=0.8,
                                        alternative="two.sided")
```

Here `h` stands for the difference between the two proportions that you want to have significant, `n1` stands for the size of one of the samples and the other parameters were already discussed before. The output to this command is depicted in the figure below:



```
              h = 0.2
             n1 = 300
             n2 = 567.2318
      sig.level = 0.05
          power = 0.8
    alternative = two.sided

NOTE: different sample sizes
```

It can be read off that the second sample must be of a length `n2` of at least 568 to meet the requirements specified by the remaining parameters.

Appendices

A Installation of R-base and R-libraries

Installation of R-base

Windows Under Windows operational systems R can easily be installed by downloading a Windows binary installer from the CRAN website. The installer is an executable file named R-xxx-win32.exe, where xxx stands for the version. Once downloaded, double-click the file to start the installation. The installation is a straight forward procedure.

MacIntosh Under MacIntosh operational systems one must distinguish between MacOS and MacOSX. Binary installers can be downloaded from the CRAN website.

Linux For a lot of Linux systems there are binary installers, which can be retrieved from the CRAN website. On some Linux systems R participates in the native package managements. If so, R can usually be installed via a few mouse clicks. If you have the necessary compilers installed, then you can also consider to install R to your machine from source code. The source code can be downloaded from the CRAN website. It is a so-called tarball, named R-xxx.tar.gz, where xxx stands for the version number. You must carry out the installation as root!

FreeBSD Under a FreeBSD operational system it is attractive to install R from source code via the (formidable) ports system. First make sure you have the ports system installed. If so, then start a console and change directory to the folder /usr/ports/math/R like this:

```
cd /usr/ports/math/R
```

Once there, type (as root) the command

```
make install clean
```

That's all. Installation from source code may take a lot of time (depending on your system it could take several hours). For the impatient there is also a possibility to install a precompiled binary distribution like this

```
pkg_add -r R
```

Make sure you have root privileges and an active internet connection!

The above also applies, grosso modo, to a lot of other BSD flavors.

Installation of R-libraries

R-libraries are extensions to R-base. They can be loaded into R, provided they are installed on your system. An easy and platform independent way to install R-libraries is as follows:

Step 1 Make sure you have administrator (or root) privileges.

Step 2 Start R as an administrator (or root).

Step 3 Under R type the command

```
install.packages("nameoflib")
```

where `nameoflib` stands for the name of the library, for example `car` or `pwr`. Don't forget the quotation marks! If the library `nameoflib` depends on other libraries, then it may be comfortable to run, rather than the above, the command:

```
install.packages("nameoflib",dependencies="Depends")
```

A panel will pop up and you will be asked to choose a server from which you want to download the library.

Step 4 Choose a server and click the button `OK`. The installation will start then.

Once the library has been installed, it can be loaded by passing the command

```
library(nameoflib)
```

The library and all the tools it provides are then at your disposal. Note that *loading* a library is quite different from *installing* a library!

On Windows systems the above procedure can partly be carried out by mouse clicking.
Another way to install a library is to first download the library from

- Libraries for Windows (http://cran.r-project.org/bin/windows/contrib/r-release/)
- Libraries for MacOSX (http://cran.r-project.org/bin/macosx/universal/contrib/r-release/)
- Source code of libraries for Unix (http://cran.r-project.org/src/contrib/)

Under a Windows operational system the library can then be installed by starting R as an administrator and type

```
install.packages("X:/path/to/nameoflib_xxx.zip",repos=NULL)
```

The option `repos=NULL` tells R that it should, rather than searching the web for repositories that contain the file `nameoflib_xxx.zip`, just use the file that resides on the place indicated by your path.

Under a MacIntosh operational system the library can be installed by starting R as an administrator and then type

```
install.packages("X:/path/to/nameoflib_xxx.tgz",repos=NULL)
```

Under Unix systems the library can be installed by starting R as root and then type

```
install.packages("X:/path/to/nameoflib_xxx.tar.gz",repos=NULL)
```

Under Unix systems the library can also be installed by running (as root) the following command in a console:

```
R CMD INSTALL /path/to/nameoflib_xxx.tar.gz
```

By this the library will be compiled to your machine from source code. You must have the necessary compilers installed on your system to bring this about!

Libraries can be deinstalled from your system by using the utility `remove.packages()`.

B How to make your own libraries under R?

B.1 Building an R-library under Windows

How to build a library for R under Windows? This section will hopefully lead you through the obstacles that people usually encounter.

B.1.1 Installation of compilers

In order to build (binary) libraries that can be installed under R under Windows, you need to have the necessary software installed. To this end you may down load the latest version of `Rtools` from http://cran.r-project.org/bin/windows/Rtools/. Once `Rtools` is installed you have everything on board to create your own libraries under R .

B.1.2 Compiling a binary installer for an R-library under Windows

Step 1 Load the stuff for your library into the R-workspace.

Step 2 Pass (under R) the command

```
package.skeleton(name="johnvanderbrook", list=ls())
```

By this action a folder, named `johnvanderbrook`, is created in the working directory of R. This folder is called the *skeleton* of the library that is to be built.

Step 3 The skeleton contains the files `DESCRIPTION`, `read-and-delete-me` and the subfolders `data`, `man`, `R` and `src`. The file `DESCRIPTION` must be edited by means of a text editor (for example WordPad, not in MS-Word). The editing could be something like this:

```
Package: broekjan
Type: Package
Title: What the package does (short line)
Version: 0.9
Date: 2008-05-12
Author: Wiebe Pestman
Maintainer: Who to complain to <w.r.pestman@uu.nl>
Description: More about what it does (more lines allowed)
License: GNU
```

The files in the subfolders `data` and `man` must also be edited. They contain files with names of type `blabla.Rd`. These files are meant to build the help functions and they must be edited as if they were simplified LaTeX files. Knowledge of LaTeX is not required in this.

Step 4 If your skeleton is ready, localize the folder where the executable `Rcmd.exe` resides. Move or copy your skeleton to this directory.

Step 5 Open the Windows Command Prompt and change directory to the folder where the file `Rcmd.exe` is sitting. This can be done via a command like this:

```
cd C:\Program Files\R\R-xxx\bin
```

Step 6 Once there, pass the command

```
Rcmd INSTALL --build johnvanderbrook
```

By this a (binary) installer `broekjan_0.9.zip` will be created. This file can be used to install the library.

B.1.3 How to use the newborn installer

The file `broekjan_0.9.zip` that was built in the previous subsection can be used to install the library `broekjan`. This can be done as follows:

Step 1 Start R.

Step 2 Follow the menu path

`Packages` $\longrightarrow$ `Install package(s) from local zip files...`

Step 3 Browse to the place where `broekjan_0.9.zip` resides and double-click the file. The installation should start then.

Step 4 After installation the library can be loaded into R by passing (under R) the command

```
library(broekjan)
```

B.2 Building an R-library under Unix-systems

To build an R-library under Unix-systems (for example Linux, FreeBSD), carry out the following steps:

Step 1 Load the stuff for your library into the R-workspace.

Step 2 Pass (under R) the command

```
package.skeleton(name="johnvanderbrook", list=ls())
```

By this action a folder, named `johnvanderbrook`, is created in the working directory of R. This folder is called the *skeleton* of the library that is to be built.

Step 3 The skeleton contains the files `DESCRIPTION`, `read-and-delete-me` and the subfolders `data`, `man`, `R` and `src`. The file `DESCRIPTION` must be edited by means of a text editor, for example like this

```
Package: broekjan
Type: Package
Title: What the package does (short line)
Version: 0.9
Date: 2008-05-12
Author: Wiebe Pestman
Maintainer: Who to complain to <w.r.pestman@uu.nl>
Description: More info (maybe more than one line)
License: GNU
```

The files in the subfolders `data` and `man` must also be edited. They contain files with names `blabla.Rd`. These files are meant to build the help functions and they can be edited as if they were simplified LaTeX files. Knowledge of LaTeX is not required in this.

Step 4 If your skeleton is ready, open a console, change directory to the place where your skeleton resides and pass the command

```
R CMD build johnvanderbrook
```

By this a tarball `broekjan_0.9.tar.gz` will be created. This tarball can be used to install your library under most Unix systems. Unfortunately not under MacOSX.

Step 5 To install your library, run (as root) the next command in a console

```
R CMD INSTALL /complete/path/to/broekjan_0.9.tar.gz
```

By this the library will be installed.

B.3 Building an R-library under MacOSX

Step 1 Load the stuff for your library into the R-workspace.

Step 2 Pass (under R) the command

```
package.skeleton(name="johnvanderbrook", list=ls())
```

By this action a folder, named `johnvanderbrook`, is created in the working directory of R. This folder is called the *skeleton* of the library that is to be built.

Step 3 The skeleton contains the files `DESCRIPTION`, `read-and-delete-me` and the subfolders `data`, `man`, `R` and `src`. The file `DESCRIPTION` must be edited by means of a text editor, for example like this

```
Package: broekjan
Type: Package
Title: What the package does (short line)
Version: 0.9
Date: 2008-05-12
Author: Wiebe Pestman
Maintainer: Who to complain to <w.r.pestman@uu.nl>
Description: More info (maybe more than one line)
License: GNU
```

The files in the subfolders `data` and `man` must also be edited. They contain files with names of type `blabla.Rd`. These files are meant to build the help functions and they can be edited as if they were simplified LaTeX files. Knowledge of LaTeX is not required in this.

Step 4 If your skeleton is ready, open a console, change directory to the place where your skeleton resides and pass the command

```
R CMD build -binary johnvanderbrook
```

By this a tarball `broekjan_0.9.tgz` will be created. This tarball can be used to install the library.

Step 5 To install your library, run (as root) the next command in a console

```
R CMD INSTALL /complete/path/to/broekjan_0.9.tgz
```

By this the library will be installed, at least, that is what I hope. Never done this myself...

References

[1] DALGAARD, Peter, *Introductory Statistics with R*, Springer Verlag, Berlin

Index