R course

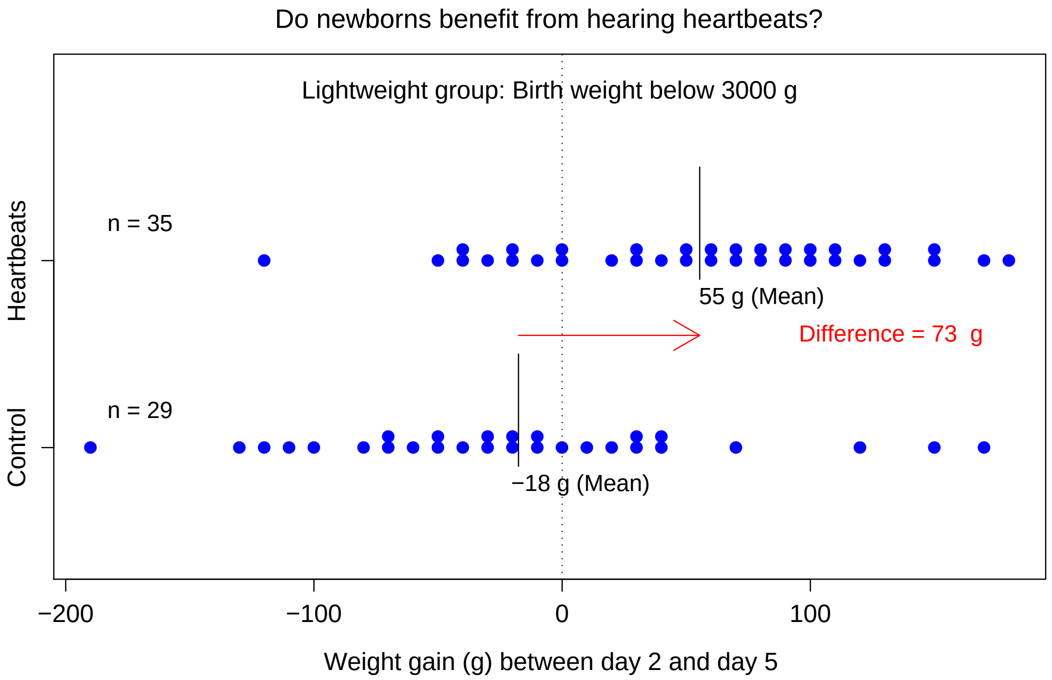
Contents

0 Getting started

0.1 What is R?

S is an environment for calculating and visualising answers to statistical questions. It is being developed since 1976 by John Chambers and colleagues at the Bell Labs, USA. *R* is an open source implementation of S. The name of R is partly based on the (first) names of the first two R authors (Robert Gentleman and Ross Ihaka, New Zealand, 1991-1993), and partly a play on the name of 'S'. In 1993 Bell Labs gave 'Insightful Corp.' (now TIBCO) an exclusive license to develop and sell the S language. Insightful sells its implementation of S under the name S-PLUS and has built a number of features (mostly GUIs) on top of it. There are only minor differences between R and S-PLUS. So most R code runs on S-PLUS and vice versa.

Here is an example of an application of R. The data is from a paper of Lee Salk (The role of the heartbeat in the relation between mother and infant, Scientific American, 1973, 228(5), 24–29). Randomly chosen newborns heard the recording of heartbeats of a grown-up. The following figure depicts the group of babies with birth weight below 3000g and compares the weight gain between day 2 and day 5 of the control group with the weight gain of the group of infants which heard heartbeats.



Main advantages:

- R is free software.
- New statistical methods are usually first implemented in R.

Disadvantages:

- R takes getting used to as it is based on command lines rather than GUIs.
- The R graphics is not interactive (e.g. there is no 'undo')

0.2 Downloading R

R is free software (GNU general public license).

Windows users: Download the R Windows installer from

http://cran.r-project.org/bin/windows/base/

Then double-click on the installer to install R as you would any Windows software. You can subsequently download and install packages you want from CRAN, via the > *Packages* > *Install packages* from *CRAN* menu in the *RGui* console. Likewise, the installer for the Tinn-R programming editor for Windows can be downloaded from

http://sourceforge.net/projects/tinn-r/

Start R by executing 'R.exe'.

Mac users: A universal binary for Mac OS X 10.4.4 and higher is available from

http://cran.r-project.org/bin/macosx/

Double-click on the icon for R.mpkg in the disk image to install R. You can then download and install packages over the Internet via the > *Packages and Data* > *Packages Installer* menu.

Linux/Unix users: Precompiled binaries for many Linux systems are available from

http://cran.r-project.org/bin/linux/

or users can compile R from source. See

http://cran.r-project.org/

for details.

0.3 Literature

- Ligges, Uwe. Programmieren mit R. SpringerVerlag Berlin 2005 (German)
- The R project web page provides the following introduction to R

  http://cran.r-project.org/doc/manuals/R-intro.pdf

- Here is a list of frequently asked questions

  http://cran.r-project.org/doc/manuals/R-FAQ.html

- Index of R commands with explanations (German, not complete yet)

  http://de.wikibooks.org/wiki/GNU_R:_Befehle-Index

- Short list of important commands

  http://cran.r-project.org/doc/contrib/Short-refcard.pdf

  This list is quite useful for reminding oneself of commands.

0.4 Libraries

There are too many R commands to load them into the memory when an R session starts. Therefore R commands are organised as packages. For example, the package 'base' contains basic commands, the package 'stats' contains statistic commands such as distributions, the package 'datasets' contains example datasets. A selection of important packages (see library(lib.loc=.Library)) is loaded at startup. Further packages need to be loaded 'by hand'. For example the package 'lattice' is loaded with

```
> library(lattice)
```

Loading a package requires the package to be installed on your computer. The list of installed packages can be viewed with installed.packages(). The list of commands contained in package *xyz* can be viewed with library(help="*xyz*"). The commands available.packages(), download.packages(), install.packages(), update.packages() might help you to install or update a package.

1 Basics

1.1 R as calculator

All beginnings are difficult. So we begin with something familiar.

```
> 2+3
[1] 5
> 3*6
[1] 18
> # This is a comment
> 2^6; 7/3         # several commands are separated with ’;’
[1] 64
[1] 2.333333
> 4+3*5^2          # precedence rules as usual
[1] 79
> 1.2
[1] 1.2
> 1,2              # German decimal notation does not work
Error: Unexpected ’,’ in "1,"
> 1.2e3            # is equal to 1.2 * 10^3. ’e’ as in ’exponent’
[1] 1200
> 5 %/% 3          # integer division
[1] 1
> 5 %% 3           # modulo division
[1] 2
> exp(1)           # exponential function
[1] 2.718282
> exp(log(5))      # log() is the natural logarithm
[1] 5
> sin(pi/2)        # sine
[1] 1
> cos(pi/2)        # cosine of pi/2 is zero. Note: R does not answer with zero!
[1] 6.123234e-17
> max(4,2,5,1)     # maximum of all elements
[1] 5
> sum(4,2,5,1)     # sum of all elements
[1] 12
> prod(4,2,5,1)    # product of all elements
[1] 40
> factorial(4)     # 4 factorial
[1] 24
> choose(5,2)      # 5 choose 2
[1] 10
# Further functions:
# exp(), log(), log10(), log2(),
# sin(), cos(), tan(), asin(), acos(), atan(),
# sinh(), cosh(), tanh(), asinh(), acosh(), atanh()
# sum(), prod(), abs(), sqrt(), max(), min(), factorial(), choose()
# round(), floor(), ceiling(), trunc(), signif()
```

1.2 Getting help

```
> help(solve)          # Displays an help page for the command "solve".
> ?solve               # Same as help(solve).
> help("exp")
> help.start()         # Shows an html-page with various links.
> help.search("solve") # Displays a list of commands which could be related to
                       # the string "solve".
> ??solve              # Same as help.search("solve").
> example(exp)         # Displays examples for the usage of 'exp'.
> example("*")         # Note that special characters have to be passed within
                       # quotation marks.
```

1.3 Assignments, comparisons and logical expressions

```
> x <- 5            # The variable x is assigned the value 5
> x
[1] 5
> 6 -> x            # equivalent to x <- 6 but unusual
> x
[1] 6
> x = 7             # equivalent to x <- 7 but unusual
> x
[1] 7
> y <- x^2 + 3      # assign 7^2 + 3 to the variable y
> y
[1] 52
> myfunction <- exp # assignment of a function
> myfunction(log(5))
[1] 5
> myfunction <- sqrt
> myfunction(81)
[1] 9
```

Subsets of large data sets are accessed by selecting all elements which have certain 'properties'. 'Properties' are formulated with logical expressions which we now introduce.

```
> 4 == 4            # Are both sides equal?
[1] TRUE            # TRUE is an R constant
> 4 == 5
[1] FALSE           # FALSE is an R constant
> 4 == 3 + 1
[1] TRUE
> cos(pi/2) == 0    # WARNING: Never compare two numerical values with ==
                    # Instead check whether the absolute value of cos(pi/2) is
                    # below a sufficiently small threshold
[1] FALSE
> 3 <= 4
[1] TRUE
> 5 > 2*2
[1] TRUE
```

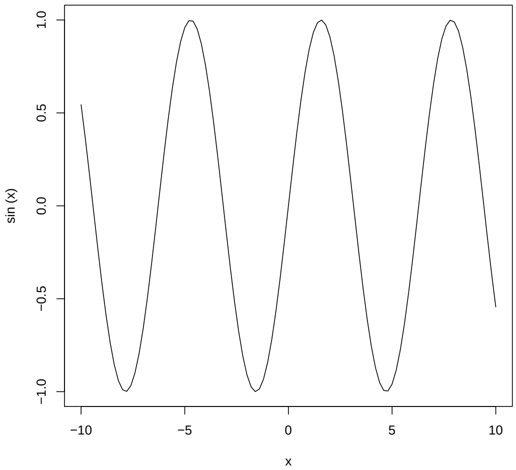
```
> 5 > 2+3
[1] FALSE
> 4 != 3            # ! is negation, != is 'not equal'
[1] TRUE
> 3 != 3
[1] FALSE
> 2 != 3
[1] TRUE
> TRUE & TRUE       # & is the logical AND. Result is TRUE if
[1] TRUE            # both expressions are TRUE
> TRUE & FALSE
[1] FALSE
> TRUE | FALSE      # | is the logical OR. Result is TRUE if
[1] TRUE            # at least one of the two expressions is TRUE
> FALSE | FALSE
[1] FALSE
> FALSE | TRUE
[1] TRUE
> 5 > 3 & 0 != 1
[1] TRUE
> 5 > 3 & 0 != 0
[1] FALSE
> ! TRUE            # ! is the logical NOT. The result of !(expression) is TRUE
[1] FALSE           # if and only if 'expression' is FALSE
> ! FALSE
[1] TRUE
> ! ( 5 == 5 )
[1] FALSE
```

1.4 Printing and plotting

Objects are printed with print(). In the interactive mode of R (you see the prompt '>'), '> print(x)' is the same as '> x'. The command cat() con'cat'enates its arguments into a single character string and prints it. The command plot() plots functions and vectors.

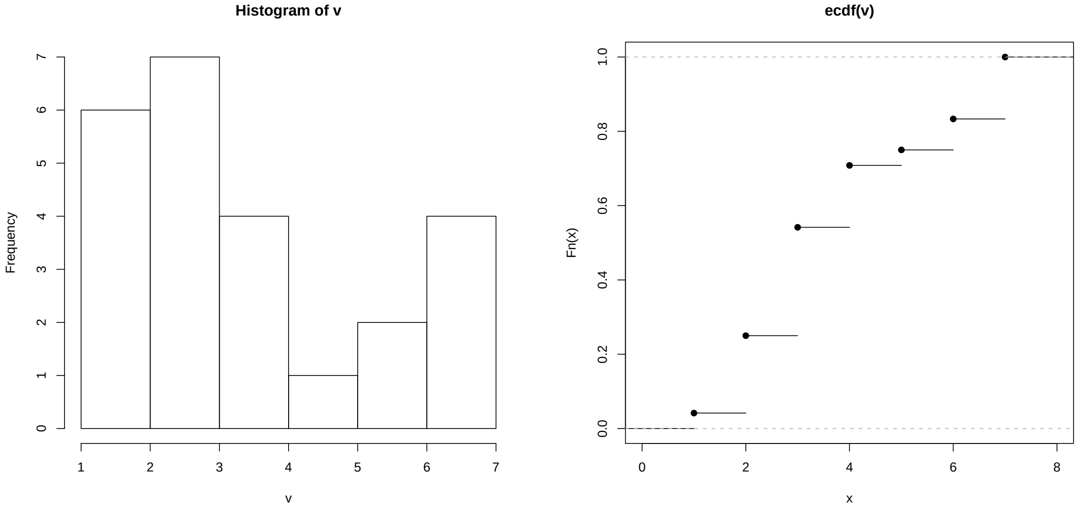
```
> print(sqrt(2))            # the same as '> x'(in the interactive mode)
[1] 1.414214
> print(sqrt(2),digits=5)   # print 5 digits
[1] 1.4142                  # see ?format for more on formatting
> sqrt(2)
[1] 1.414214                # the default is to print 7 digits
> options(digits=10)        # 10 digits are printed from now on
> sqrt(2)
[1] 1.414213562
> y <- 42
> cat("And the answer is",y,"as I believe.\n")  # \n produces a new line
And the answer is 42 as I believe.
> cat("Today is:\t",date(),"\n")       # date() returns today's date and time
Today is:       Fri Jan 01 06:00:00 2010
> plot(sin, from=-10, to=10)
```

Here we used `'\n'` for a new line and `'\t'` for inserting a tab – see `?Quotes` for a complete list.



The histogram of a vector v is plotted with hist(v). The command ecdf(v) returns the empirical cumulative distribution function ('ecdf') of the vector v.

```
> v <- c ( 1:7, rep(2,4), rep(3,6), rep(4,3), 6:7, 7, 7 )
> hist(v)
> ecdf(v)       # ecdf() returns the ecdf and does not print it
Empirical CDF
Call: ecdf(v)
 x[1:7] =      1,      2,      3,  ...,      6,      7
> plot(ecdf(v))
```



1.5 Vectors

Vectors are enumerations of arbitrary objects. The commands 'c', 'seq' and 'rep' create vectors.

```
> c(2,5,3,7)             # concatenate elements into a vector
[1] 2 5 3 7
> seq(from=1,to=10,by=3) # create a sequence from 1 to 10 with step size 3
[1]  1  4  7 10
> seq(from=3,to=7)       # short for seq(from=3,to=7,by=1)
[1] 3 4 5 6 7
> seq(1,11,3)            # short for seq(from=1,to=11,by=3).
                         # Here the order of 1, 11 and 3 matters!
[1]  1  4  7 10
> seq(3,7)               # short for seq(from=3,to=7). Again order matters!
[1] 3 4 5 6 7
> 3:7                    # short for seq(from=3,to=7)
[1] 3 4 5 6 7
> c(2:5, 3:7)
[1] 2 3 4 5 3 4 5 6 7
> rep(3,5)               # replicate first object (here 3) 5 times
[1] 3 3 3 3 3
> rep(0:2,3)
[1] 0 1 2 0 1 2 0 1 2
> rep(7:9,2:4)           # If both arguments are vectors with the same length.
[1] 7 7 8 8 8 9 9 9 9    # Here: Replicate 7 two times, 8 three times, 9 four
                         # times
```

You access elements of a vector with the [ ]-Operator.

```
> x <- c(12,15,13,17,11)
> x[4]                   # The fourth element of the vector x
[1] 17
> x[3:5]                 # subvector with indices 3, 4 and 5
[1] 13 17 11
> x[-2]                  # the minus means ’without’
[1] 12 13 17 11
> x[-(3:5)]              # all elements except 3,4 and 5
[1] 12 15
```

Standard operations on vectors are element by element. In a binary operation, the length of one vector has to be a multiple of the length of the other vector. Examples:

```
> c(2,5,3) + c(4,2,7)
[1]  6  7 10
> c(2,5,3) * c(4,2,7)
[1]  8 10 21
> 2 + c(2,5,3)                   # same as c(2,2,2) + c(2,5,3)
[1] 4 7 5
> 2 * c(2,5,3)                   # same as c(2,2,2) * c(2,5,3)
[1]  4 10  6
> c(2,5,3)^2                     # same as c(2,5,3)^c(2,2,2)
[1]  4 25  9
> c(2,5,3)^c(3,2,1)
[1]  8 25  3
> c(3,2) * c(2,5,3,4)            # same as c(3,2,3,2) * c(2,5,3,4)
```

```
[1]  6 10  9  8
> c(2,5,3,4)^c(3,2)              # same as c(2,5,3,4)^c(3,2,3,2)
[1]  8 25 27 16
> c(3,2)^c(2,5,3,4)
[1]  9 32 27 16
> exp( c(0,1,log(5)) )
[1] 1.000000 2.718282 5.000000
> sum(5:7)                       # 5 + 6 + 7
[1] 18
> prod(4:6)                      # 4 * 5 * 6
[1] 120
> x <- 1:5
> x > 3                           # which elements are > 3
[1] FALSE FALSE FALSE  TRUE  TRUE
> (x %% 2) == 1                   # which elements are odd
[1]  TRUE FALSE  TRUE FALSE  TRUE
```

A strong feature of R is indexing with logical index vectors. Thereby we may select the subvector consisting of all elements with certain properties.

```
> v <- c(12,15,13,17,11)
> v[c(TRUE,FALSE,TRUE,TRUE,FALSE)]
[1] 12 13 17
> v[c(TRUE,FALSE,TRUE)]      # shorter index vectors are filled up with 'FALSE'
[1] 12 13
> v[c(TRUE,FALSE,TRUE,TRUE,FALSE,TRUE,TRUE)]
[1] 12 13 17 NA NA
> v>12
[1] FALSE  TRUE  TRUE  TRUE FALSE
> v[v>12]
[1] 15 13 17
>  v<=16 & (v%%2)==1
[1] FALSE  TRUE  TRUE FALSE  TRUE
> v[ v<=16 & (v%%2)==1 ]
[1] 15 13 11
> v[ v>=13 | (v%%2)==0 ]
[1] 12 15 13 17
> v[v>12] <- 0               # set all elements which are bigger than 12 to 0
> v
[1] 12  0  0  0 11
> v[v==0] <- 2               # assign 2 to all elements being equal to 0
> v
[1] 12  2  2  2 11
> v==2
[1] FALSE  TRUE  TRUE  TRUE FALSE
```

Here is a selection of commands on vectors.

```
> v <- c(13, 15, 11, 12, 19, 11, 17, 19)
> length(v)                  # returns the length of the vector v
[1] 8
```

```
> rev(v)                      # returns the ’rev’ersed vector
[1] 19 17 11 19 12 11 15 13
> sort(v)                     # returns the sorted vector
[1] 11 11 12 13 15 17 19 19
> sort(v,partial=4)           # returns a vector in which the fourth element
[1] 11 11 12 13 19 15 17 19  # agrees with the sorted vector
> indexvec <- order(v)
> indexvec                    # the index vector for sorting
[1] 3 6 4 1 2 7 5 8
> v[indexvec]                 # v[ order(v) ] is the same as sort(v)
[1] 11 11 12 13 15 17 19 19
> duplicated(v)               # identifies multiple elements
[1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
> unique(v)                   # returns vector without multiple elements
[1] 13 15 11 12 19 17
> rank(v)                     # returns the ranks of the elements in vector v
[1] 4.0 5.0 1.5 3.0 7.5 1.5 6.0 7.5
> some <- ( v > 13 )
> some
[1] FALSE  TRUE FALSE FALSE  TRUE  FALSE TRUE  TRUE
> any(some)    # returns TRUE if at least one entry is TRUE; else FALSE
[1] TRUE
> all(some)    # returns TRUE if all entries of ’some’ are TRUE; else FALSE
[1] FALSE
> which(some)  # returns the indices at which ’some’ is TRUE
[1] 2 5 7 8
> which.max(v) # returns the index of the maximum (first such index)
[1] 5
> which.min(v) # returns the index of the minimum (first such index)
[1] 3
```

1.6 Matrices

Matrices are usually created with ’matrix’, by converting a vector into a matrix or by binding together vectors.

```
> m <- matrix( data = 1:8, nrow=4, ncol=2 )
> m
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
> matrix ( 1:8,4,2 )      # Same as matrix( data = 1:8, nrow=4, ncol=2 )
```

Indexing is ’row by column’.

```
> m[3,2]                  # Entry in the third row and second column.
[1] 7
> m[2,]                   # Second row
```

```
[1] 2 6
> m[,2]                   # Second column
[1] 5 6 7 8
> m[2:3,1:2]              # submatrix
     [,1] [,2]
[1,]    2    6
[2,]    3    7
```

The command 'matrix' stores the vector of data by default column by column, that is, it starts filling the first column first. If you wish to specify the data row by row, then add 'byrow = TRUE'.

```
> y <- matrix( data = 1:8, nrow=4, ncol=2, byrow = TRUE )
> y
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
> t(y)                    # t(y) is the transposed matrix of y
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> z <- as.matrix(1:6)     # Convert the vector 1:6 into a matrix with one column
> z
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
[5,]    5
[6,]    6
> dim(z)
[1] 6 1
> dim(z) <- c(2,3)        # Convert the 6 by 1 matrix z into a 2 by 3 matrix
> z
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> dim(z)
[1] 2 3
> is.matrix( 1:6 )
[1] FALSE
> is.matrix( as.matrix(1:6) )
[1] TRUE
> cbind(1:3,5:7)          # Bind together vectors column-wise
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
> rbind(1:3,5:7,10:12)    # Bind together vectors row-wise
     [,1] [,2] [,3]
```

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

The command 'diag' creates diagonal matrices.

```
> diag(1,nrow=3,ncol=3)  # Creates the 3 by 3 unit matrix.
     [,1] [,2] [,3]      # Diagonal matrix with 3 rows and 3 columns
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
> diag(3)                # same as diag(1,nrow=3,ncol=3)
> diag(5:7,3,4)
     [,1] [,2] [,3] [,4]
[1,]    5    0    0    0
[2,]    0    6    0    0
[3,]    0    0    7    0
```

As for vectors, standard operations on matrices are element by element.

```
> m1 <- matrix(1:6,nrow=3) ; m1
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> m2 <- matrix(16:11,nrow=3) ; m2
     [,1] [,2]
[1,]   16   13
[2,]   15   12
[3,]   14   11
> m1 + m2
     [,1] [,2]
[1,]   17   17
[2,]   17   17
[3,]   17   17
> m1 * m2
     [,1] [,2]
[1,]   16   52
[2,]   30   60
[3,]   42   66
```

Note that m1$*$m2 is not the matrix product. The operator for matrix multiplication is '%*%'. The product of a m by n matrix y and of a k by l matrix z is only defined if $n == k$. By definition, the (i, j)-entry of $y \,\%*\%\, z$ is $\sum_{a=1}^{n} y[i, n] * z[n, j]$.

```
> mat1 <- matrix(1:6,2,3)
> mat2 <- matrix(5:0,2,3)
> mat1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> mat2
```

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

In a matrix product between a vector and a matrix, the vector is interpreted as row vector for 'vector %*% matrix' and is interpreted as column vector for 'matrix %*% vector'. The convention for 'vector %*% vector' is to interprete the product as scalar product 'rowvector %*% columnvector'.

```
> mat3 <- t(mat2) ; mat3
     [,1] [,2]
[1,]    5    4
[2,]    3    2
[3,]    1    0
> mat1 %*% mat3        # 2 by 3 matrix  times  3 by 2 matrix
     [,1] [,2]
[1,]   19   10
[2,]   28   16
> mat3 %*% mat1        # 3 by 2 matrix  times  2 by 3 matrix
     [,1] [,2] [,3]
[1,]   13   31   49
[2,]    7   17   27
[3,]    1    3    5
> v <- 7:9
> v %*% mat3           # v is interpreted as row vector
     [,1] [,2]
[1,]   14    8
> v %*% v              # scalar product of v with itself
     [,1]
[1,]  194
# column vector %*% row vector has to be enforced
> as.matrix(v)%*% t( as.matrix(v) )
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    4    6
[3,]    3    6    9
```

Note that vectors are neither always interpreted as row vectors nor are vectors always interpreted as column vectors. The interpretation depends on the evaluating function.

1.7 Data types in R

Every variable and object in R has a class (e.g. matrix, list, data frame) and a data type (e.g. logical, numeric, complex, character) which species what type of data the object contains.
Data types in R:

Data type	**Discreption**	**Examples**
logical	TRUE or FALSE	TRUE, FALSE
numeric	integers and real numbers	5, -2, 3.1415, sqrt(2)
complex	complex numbers	2.1+3i, 5+0i
character	character string	"This is text", "5"

Types can be converted with the commands `as.logical()`, `as.numeric()`, `as.complex()` and `as.character()`. Conversion (also called coercion) of data types is usually not done by the user but implicitly by R where necessary. Note that implicit conversion only goes in the direction logical → numeric → complex → character. You check for the data type of a variable with the commands `is.logical()`, `is.numeric()`, `is.complex()`, `is.character()`. You find out the data type of a variable with `mode()` and the class of a variable with `class()`.

Examples:

```
> x <- TRUE; mode(x)
[1] "logical"
> as.numeric(TRUE) ; as.numeric(FALSE)
[1] 1               # the internal representation of TRUE is 1
[1] 0               # the internal representation of FALSE is 0
> c("1.0",2)        # the 2 is implicitly coerced to 'character'
[1] "1.0" "2"
> v <- c(1,2,2,2,5); as.numeric(v==2)
[1] 0 1 1 1 0
> sum(v==2)                # same as sum( as.integer(v==2) )
> [1] 3                    # How many elements are equal to 2?
> 2*c(TRUE, FALSE)         # Implicit coercion logical -> numeric works
[1] 2 0
> as.numeric(c("1.0", "2")) # Explicit coercion character -> numeric if possible
[1] 1 2
> 2*c("1.0","2")           # Implicit coercion logical -> numeric does not work
Error in 2 * c("1.0", "2") : non-numeric argument to binary operator
```

2 Basic Statistics with R

2.1 Some distributions implemented in R

R provides the following distributions:

Distribution	**R name**
beta distribution	beta
binomial distribution	binom
Cauchy distribution	cauchy
chi-square distribution	chisq
exponential distribution	exp
F-distribution	f
gamma distribution	gamma
geometric distribution	geom
hypergeometric distribution	hyper
log-normal distribution	lnorm
logistic distribution	logis
multinomial distribution	multinom
multivariate normal distribution	mvnorm
negative binomial distribution	nbinom
normal distribution	norm
Poisson distribution	pois
distribution of the Wilcoxon signed rank statistic	signrank
student's t-distribution	t
uniform distribution	unif
Weibull distribution	weibull
distribution of the Wilcoxon rank sum statistic	wilcox

The normal distribution is the most important distribution due to the central limit theorem.The binomial distribution and the uniform distribution are certainly good to know. The chi-square distribution, student's t-distribution and the F-distribution are used in standard tests.All other distributions will not turn up in the course again.

For each distribution, R provides the following four commands:

dxxx:	density of the xxx distribution
pxxx:	distribution function of the xxx distribution ('p' for probability)
qxxx:	quantile function of the xxx distribution
rxxx:	random number generator for the xxx distribution

Just replace 'xxx' by the R name of the distribution. For example, dnorm is the density of the normal distribution.

The density function of the **normal distribution** with mean $\mu \in \mathbb{R}$ and standard deviation $\sigma \in (0, \infty)$ is

$$\mathrm{dnorm}(x, \mathrm{mean} = \mu, \mathrm{sd} = \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\Big(-\frac{(x-\mu)^2}{2\sigma^2}\Big), \quad x \in \mathbb{R}.$$

Its mean is μ and its variance is σ^2. The standard normal distribution has mean 0 and standard deviation 1. So its density is $\mathrm{dnorm}(x, \mathrm{mean} = 0, \mathrm{sd} = 1)$. If 'mean' and 'sd' are not specified in dnorm() they assume the default values 0 and 1, respectively. So dnorm(x) is the same as dnorm(x, mean $= 0$, sd $= 1$). The distribution function satisfies

$$\text{pnorm}(x) <\!- \int_{-\infty}^{x} \text{dnorm(y)}\; dy, \quad x \in \mathbb{R}.$$

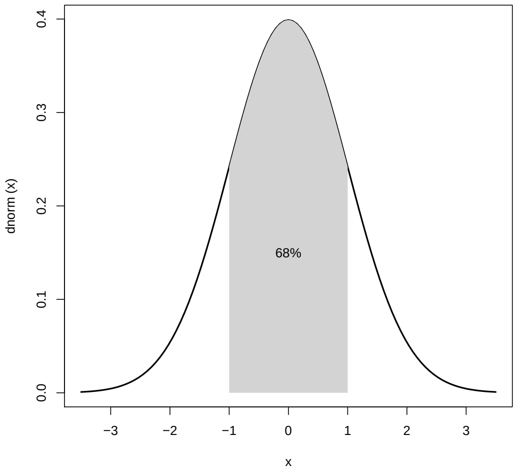
The quantile function at $q \in [0, 1]$ is the smallest value x such that pnorm$(x) \geq q$. More formally, we have

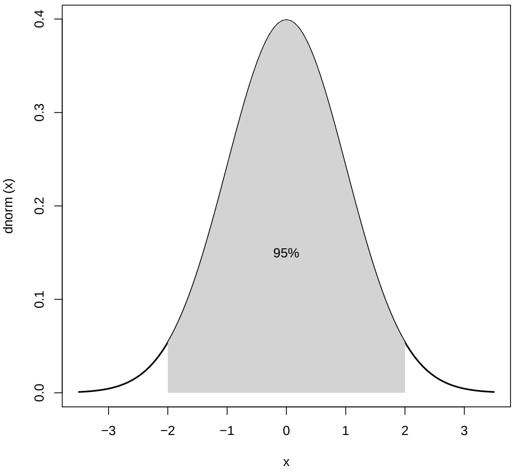
$$\text{qnorm}(q) <\!- \min\Big(x \in \mathbb{R}\colon\ \text{pnorm}(x) \geq q\Big), \quad q \in [0, 1].$$

Note that if the distribution function is continuous (as pnorm is), then the quantile function is the inverse function of the distribution function. The command rnorm(n) generates a random sample of length n of the normal distribution, that is, it produces n random values which are independent and whose distribution is standard normal.

The following facts are useful to remember: 68% of the mass of a standard normal distribution is within one standard deviation; 95% of the mass is within two standard deviations.

```
> pnorm(1)-pnorm(-1)      # 68 % is within one standard deviation
[1] 0.6826895
> pnorm(2)-pnorm(-2)      # 95 % is within two standard deviations
[1] 0.9544997
> pnorm(3)-pnorm(-3)      # 99.7 % is within 3 standard deviations
[1] 0.9973002
```



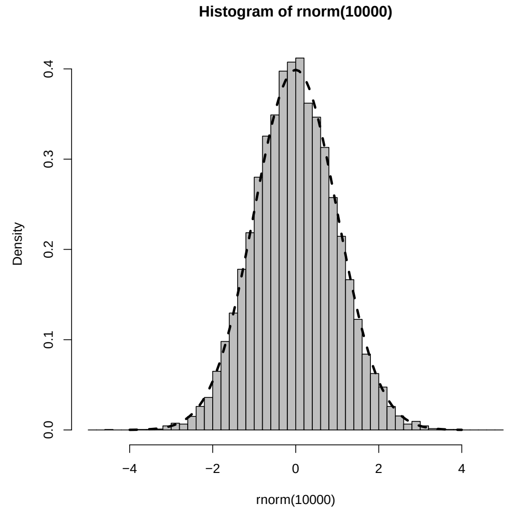


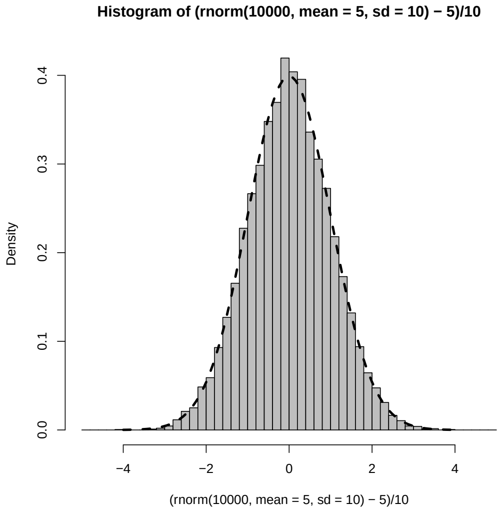
Another property to remember is the following scaling property of the normal distribution. If the distribution of $X_{0,1}$ is standard normal distributed, then the distribution of $X_{\mu,\sigma} := \sigma X_{0,1} + \mu$ is normal with mean μ and variance σ^2. If the distribution of $X_{\mu,\sigma}$ is normal with mean μ and variance σ^2, then $X_{0,1} := \frac{X_{\mu,\sigma}-\mu}{\sigma}$ is standard normally distributed.

```
> # The histogram of 10000 simulated values is close to the density function:
> hist(rnorm(10000),col="grey",probability=TRUE,breaks=seq(-5,to=5,by=0.2))
> plot( dnorm, from=-4,to=4,add=TRUE ,lwd=3, lty="dashed")
```

The last command plots `dnorm` between -4 and 4 with line width 3 and line type `"dashed"`.

```
> # The following example shows that a non-standard normal random variable
> # suitably rescaled is a standard normal variable:
> hist( (rnorm(10000,mean=5,sd=10)-5) / 10,col="grey",probability=TRUE,
+ breaks=seq(-5,to=5,by=0.2) )
> plot( dnorm, from=-4,to=4,add=TRUE ,lwd=3, lty="dashed")
```





The **binomial distribution** is the distribution of the

number of 'successes' in a row of independent trials.

The density dbinom(k, m, p) is the probability to have $k \in \{0, \dots, m\}$ 'successes' in a row of m trials with 'success' probability $p \in [0, 1]$. The density function of the binomial distribution satisfies

$$\text{dbinom}(k, m, p) <\!- \binom{m}{k} p^k \cdot (1-p)^{m-k}.$$

The mean is $m \cdot p$ and the variance is $m \cdot p \cdot (1 - p)$. The distribution function of the binomial distribution satisfies

$$\text{pbinom}(k, m, p) <\!- \sum_{l=0}^{k} \text{dbinom}(l, m, p)$$

The quantile function qbinom(q, m, p) at $q \in [0, 1]$ is the smallest value x such that dbinom(x, m, p) is bigger than or equal to q. The command rbinom(n, m, p) generates a random sample of length n.

```
> dbinom(1,3,1/6)           # Probability of one 6 when throwing a dice 3 times
[1] 0.3472222
> dbinom(0,1,1/2)           # Probability of heads when throwing one coin
[1] 0.5
> dbinom(1,1,1/2)           # Probability of tails when throwing one coin
[1] 0.5
> sum(dbinom(5:10,60,1/6)) # Probability to have between five and ten 6's
[1] 0.5631944               # when throwing a dice 60 times.
> sum(dbinom(0:60,60,1/6)) # The total probability is equal to 1
```

```
[1] 1
> qbinom(0.5833866,60,1/6)
[1] 11
> qbinom(0.5833865,60,1/6) # qbinom is basically the inverse function of pbinom
[1] 10
> qbinom(pbinom(0:29,60,1/6),60,1/6) == 0:29
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
16] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> rbinom(8,60,1/6)
[1] 12 10 11 11  4 12  7 11
```

In the last example, each of the 8 numbers is the number of 6's after throwing a dice 60 times.

The **uniform distribution** on the interval $[a, b]$, $a < b$, has density

$$\text{dunif}(x, \text{min} = a, \text{max} = b) <- \frac{1}{b-a} \qquad \text{for all } x \in [a, b].$$

The standard uniform distribution is on the interval $[0, 1]$. So its density is $\text{dunif}(x, \text{min} = 0, \text{max} = 1)$. If 'min' and 'max' are not specified in dnorm() they assume the default values 0 and 1, respectively. So $\text{dunif}(x)$ is the same as $\text{dunif}(x, \text{min} = 0, \text{max} = 1)$. The distribution function satisfies

$$\text{punif}(x) <- \int_0^x \text{dunif(y)}\ dy = x \qquad \text{for all } x \in [0, 1].$$

The mean of the standard uniform distribution is $\frac{1}{2}$ and the variance is $\frac{1}{12}$.

2.2 Examining the distribution of a set of data

mean(v) Computes the sample mean $\frac{1}{n}\sum_{i=1}^{n} v_i$ of the vector v where $n = \text{length}(v)$.

var(v) Computes the sample variance $\frac{1}{n-1}\sum_{i=1}^{n}\big(v_i - \text{mean}(v)\big)^2$ of the vector v where $n = \text{length}(v)$.

sd(v) Computes the sample standard deviation $\sqrt{\text{var}(v)}$ of the vector v where $n = \text{length}(v)$.

cov(v, w) Computes the sample covariance $\frac{1}{n-1}\sum_{i=1}^{n}\big(v_i - \text{mean}(v)\big)\big(w_i - \text{mean}(w)\big)$ of the two vectors v and w.

cor(v, w) Computes the sample correlation $\text{cov}(v, w)/\sqrt{\text{var}(v)\,\text{var}(w)}$ of the two vectors v and w.

median(v) A median is a real number such that half of the vector elements are below this number and half of the vector elements are above this number. The median is usually not unique. R returns the average of all medians.

quantile(v) quantile() applied to a numeric vector v displays by default the quartiles of v.

summary(v) summary() applied to a numeric vector v displays mean, variance and median of v together with the quartiles.

boxplot(v) Box- and whisker plot

hist(v) hist() plots the histogram of v.

ecdf(v) Returns the empirical cumulative distribution function of v.

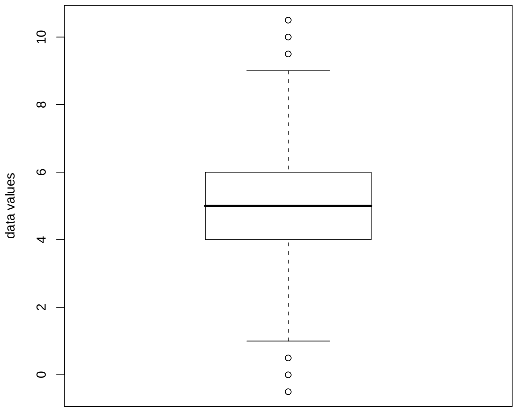
Let us recall what the quantile function, the median and the quartiles are. Let $p \in [0, 1]$. Below the p-quantile is a fraction p of all values of the vector. For more details, see ?quantile. Any 0.25-quantile is referred to as *first quartile*, any 0.5- quantile is referred to as *median* (or *second* *quartile*) and any 0.75-quantile is referred to as *third quartile*. So the quartiles split the vector into quarters.

A fraction of 25% of the elements lie below the first quartile and 25% of the elements lie above the third quartile. The middle 50% of the data lie between the first and the third quartile.

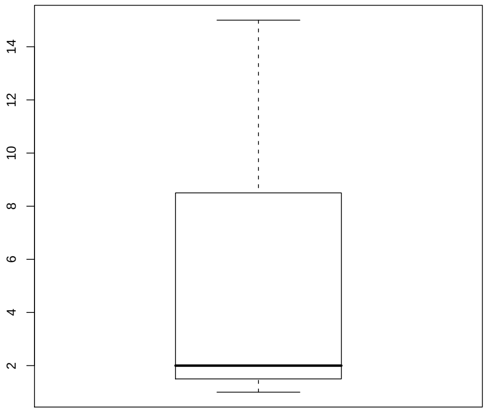
Here are some examples:

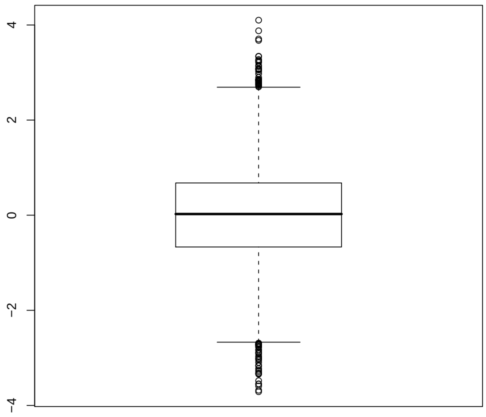
```
> mean(c(1,2,21))     # (1+2+21)/3
[1] 8
> mean(c(0,1,2,21))   # (0+1+2+21)/4
[1] 6
> mean(c(-1,0,1))
[1] 0
> var(c(-1,0,1))      # ( (-1)^2+1^2 ) / ( 3-1 )
[1] 1
> var(c(-2,0,2))
[1] 4
> var(rep(1,4))
[1] 0
> mean(rep(1,4))
[1] 1
> mean(1:4)           # ( 1+2+3+4 ) / 4
[1] 2.5
> cov(1:4,1:4)        # same as var(1:4)
[1] 1.666667          # ( (1-2.5)^2+(2-2.5)^2+(3-2.5)^2+(4-2.5)^2 ) / ( 4-1 )
> cor(1:4,1:4)        # = cov(1:4,1:4) / sqrt( var(1:4)*var(1:4) )
[1] 1
> cor(1:4,-1:4)
[1] -1
> cor(1:4,17:20)
[1] 1
> cor(1:4,20:17)
[1] -1
> median(c(1,2,9,1100))
[1] 5.5
> quantile(c(1,2,9,1100))
     0%     25%     50%     75%    100%
   1.00    1.75    5.50  281.75 1100.00
> summary(c(1,2,15))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0     1.5     2.0     6.0     8.5    15.0
> mean(rnorm(1000000))
[1] 0.0004818741
```

A box- and whisker plot (short boxplot) visualizes the quartiles. The horizontal bar in the middle shows the median value of the data. The horizontal line above the median shows the third quartile and the horizontal line below the median shows the first quartile. The box shows where the middle 50% of the data lie (this is called 'the interquartile range'). The length of the interquartile range is called interquartile distance. The length of each of the two whiskers (German 'Barthaare') is limited by 1.5 times the interquartile distance. In fact they reach to the maximum/minimum point within that distance. All data points outside the whiskers are called *outliers* and are indicated by single points in the boxplot

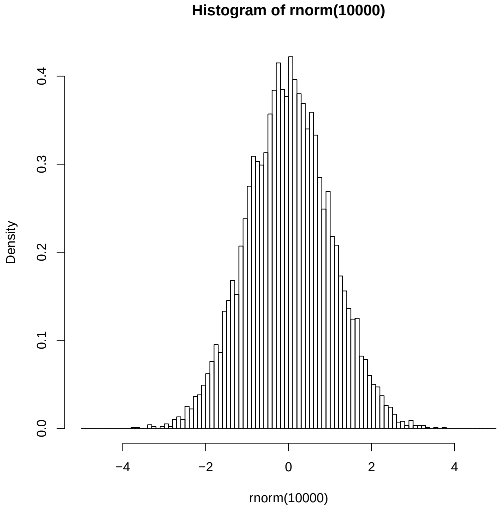


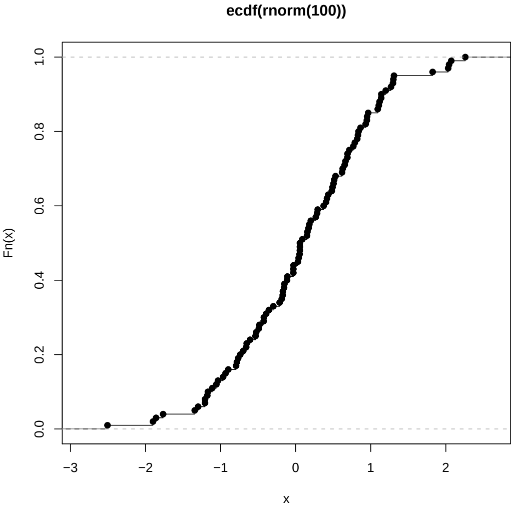
```
> summary(c(1,2,1500))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0     1.5     2.0   501.0   751.0  1500.0
> boxplot(c(1,2,1500))
> summary(rnorm(10000))
   Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-3.858000 -0.690200  0.005504 -0.003528  0.667200  3.944000
> boxplot(rnorm(10000))
> summary(rbinom(10000,60,1/6))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    8.00   10.00   10.01   12.00   22.00
> summary(runif(1000000))
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
1.036e-06 2.496e-01 5.002e-01 5.000e-01 7.506e-01 1.000e+00
```





```
> hist(rnorm(10000))
> plot(ecdf(rnorm(100)))
```





2.3 Random number generators

The random numbers produced by R are in fact only pseudo-random numbers. There are several reasons for this. First of all, a normally distributed variable has a continuum (ℝ) of potential values. Computers, however, can only represent a finite number of values. The second reason is the desire to have reproducible results. Pseudo-random numbers are supposed to have the following properties:

- (almost) no regularities in the generated sequence
- random sequence is reproducible

- random sequence is generated as quick as possible.

The pseudo-random number generator is initialised with the so-called *seed* using the common set.seed(). The sequence is then generated by applying a certain deterministic function again and again. Thereby if the generator is initialised with the same seed again, then it generates the same sequence. The default generator in R is *Mersenne-Twister* (Matsumoto and Nishimura 1998). You can change the kind of random number generator with the command RNGkind(), see ?RNGkind. Note that the random number generator is initialised using the system time at the beginning of the R session. If you wish to reproduce your results, then use *set.seed()*.

```
> rnorm(3)
[1]  1.0844412 -2.3456977  0.4291247
> set.seed(1234)                 # initialise random number generator
> rnorm(3)
[1] -1.2070657  0.2774292  1.0844412
> rnorm(3)
[1] -2.3456977  0.4291247  0.5060559
> set.seed(1234)                 # restart random number generator
> rnorm(3)                       # same values as before
[1] -1.2070657  0.2774292  1.0844412
> rnorm(3)
[1] -2.3456977  0.4291247  0.5060559
> RNGkind("Wichmann-Hill")       # different kind of random number generator
> set.seed(1234)                 # initialise with same seed
> rnorm(3)                       # values are different due to different RNG
[1] -0.2160838  0.8444022  0.6975076
> RNGkind("Mersenne-Twister")
> set.seed(1234)
> rnorm(3)
[1] -1.2070657  0.2774292  1.0844412
```

3 Reading and writing data

3.1 Lists

Data sets are represented as so-called data frames in R. These are special lists which we now introduce. Lists are collections of arbitrary objects. They are created with the list() command. The elements of the list are accessed with the [[ ]]-operator.

```
> L <- list( c(1,5,3), matrix(1:6, nrow=3), c("Hello", "world") )
> L
[[1]]
[1] 1 5 3

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

[[3]]
```

```
[1] "Hello" "world"

> L[[1]]          # First element of L
[1] 1 5 3
> L[[2]][2,1]     # Element [2,1] of the second element of L
[1] 2             # Note that L[[2]] is a matrix which can be referenced with []
> L[[c(3,2)]]     # Recursively: 3. element of L, hereof the 2. element
[1] "world"
> list(1:4,7:8) # A list of two vectors
[[1]]
[1] 1 2 3 4

[[2]]
[1] 7 8

> c(1:4,7:8)      # c() concatenates the two vectors into one vector
[1] 1 2 3 4 7 8

# when concatenating lists, however, c() produces the concatenated list
> L2 <- c( list(1:3,2:4), list( c("Hello","world"), c(1,5,3) ) )
> L2
[[1]]
[1] 1 2 3

[[2]]
[1] 2 3 4

[[3]]
[1] "Hello" "world"

[[4]]
[1] 1 5 3

> mode(L2)            # mode() shows the type of the object
[1] "list"            # indeed, L2 is a list

> L2[[2]] <- NULL     # setting the second element of the list to NULL
> L2                  # is the same as deleting the second element
[[1]]
[1] 1 2 3

[[2]]
[1] "Hello" "world"

[[3]]
[1] 1 5 3
```

Lists can alternatively be referenced by name instead of by number. In the above example, one needs to remember that the matrix in L is the second element. The following definition is easier to remember as the elements can be referred to by name. The elements of the list are accessed with the $-operator.

```
> L <- list( v=c(1,5,3), m=matrix(1:6, nrow=3), text=c("Hello", "world") )
> L$v
[1] 1 5 3
> L$m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> L$text
[1] "Hello" "world"
> L$m[2,1]          # L$m is a matrix which can be referenced with []
[1] 2
> L$text[2]
[1] "world"
> L[[1]]            # referencing by number still works
[1] 1 5 3
```

3.2 Data frames

Data frames are the typical R representation of data sets. Data frames are lists with the constraint that all elements are vectors of the same length. The command data.frame() creates a data frame.

```
> group <- data.frame(name = c("Hans", "Caro", "Lars", "Ines", "Samira",
 "Peter", "Sarah"), gender = c("male", "female", "male", "female",
 "female", "male", "female"), favourite_colour = c("green", "blue", "yellow",
 "black", "yellow", "green", "black"),
 income = c(800,1233,2400,4000,2899,1100,1900) )
> group
    name      gender favourite_colour income
1   Hans        male            green    800
2   Caro      female             blue   1233
3   Lars        male           yellow   2400
4   Ines      female            black   4000
5 Samira      female           yellow   2899
6  Peter        male            green   1100
7  Sarah      female            black   1900
> group$income
[1]  800 1233 2400 4000 2899 1100 1900
> group$gender[2]
[1] female
Levels: female male
> group[,1]
    name
1   Hans
2   Caro
3   Lars
4   Ines
5 Samira
6  Peter
```

```
7  Sarah
> group$name
[1] Hans   Caro   Lars   Ines   Samira Peter  Sarah
Levels: Caro Hans Ines Lars Peter Samira Sarah
> group[1,]
  name gender favourite_colour income
1 Hans   male            green    800
```

Data frames usually accomodate large data sets which are difficult to get an overview of. The command str() gives an overview of all variables in the data frame. The command summary() prints a summary of the data frame.

```
> str(group)
'data.frame': 7 obs. of  4 variables:
 $ name            : Factor w/ 7 levels "Caro","Hans",..: 2 1 4 3 6 5 7
 $ gender          : Factor w/ 2 levels "female","male",..:3 1 2 1 1 3 1
 $ favourite_colour: Factor w/ 4 levels "black","blue",..: 3 2 4 1 4 3 1
 $ income          : num  800 1233 2400 4000 2899 ...
> summary(group)
     name          gender  favourite_colour     income
 Caro  :1   female       :4   black :2         Min.   : 800
 Hans  :1   male         :3   blue  :1         1st Qu.:1166
 Ines  :1                     green :2         Median :1900
 Lars  :1                     yellow:2         Mean   :2047
 Peter :1                                      3rd Qu.:2650
 Samira:1                                      Max.   :4000
 Sarah :1
```

Writing 'group$' all the time is tiring. To avoid this say attach(group). This command copies 'group' into the search path of R (imagine a place where R looks for variables) so that all elements can be found without mentioning 'group$'.

```
> gender[2]
Error: object "gender" not found
> attach(group)        # copy 'group' into the search path
> gender[2]            # 'gender' is now a known variable
[1] female
Levels: female male
> name[3]
[1] Lars
Levels: Caro Hans Ines Lars Peter Samira Sarah
> name[3] <- "Samira"  # this only changes the copy of group in the search path
> name
[1] Hans   Caro   Samira Ines   Samira Peter  Sarah
Levels: Caro Hans Ines Lars Peter Samira Sarah
> group$name           # the original 'group' remains unchanged
[1] Hans   Caro   Lars   Ines   Samira Peter  Sarah
Levels: Caro Hans Ines Lars Peter Samira Sarah
> detach(group)        # removes 'group' from the search path
> name[3]              # after detach(group), 'name' is no longer known
Error: object "name" not found
```

```
> gender[2]
Error: object "gender" not found
```

Looking at 'group' one might think that people with favourite colour 'yellow' have an higher income. To investigate this, we wish to select the subset of 'group' with favourite colour 'yellow'. This could be done with

```
> group[group[["favourite_colour"]]=="yellow",]
    name      gender favourite_colour income
3   Lars        male           yellow   2400
5 Samira      female           yellow   2899
```

A more convenient solution is provided by the command subset():

```
> subset(group,favourite_colour=="yellow")
```

Of course the column of the favourite colour is identically 'yellow' so we do not need it. We delete it by selecting everything except the third column.

```
> subset(group,favourite_colour=="yellow",select=-3)
    name      gender income
3   Lars        male   2400
5 Samira      female   2899
```

One could also wish to have the subset of people favouring the colours 'green' and 'black'. So favourite_colour ∈ {green, black} would be convenient. The symbol ∈ is represented in R with the operator %in%.

```
>  subset( group, favourite_colour %in% c("green","black") )
   name gender favourite_colour income
1  Hans   male            green    800
4  Ines female            black   4000
6 Peter   male            green   1100
7 Sarah female            black   1900
```

Now does the subgroup with favourite colour 'yellow' have more income? We calculate the sample mean of the 'yellow'-group income, which turns out to be higher. Of course, this does not answer the question as the sample size is way too small to have a significant result.

```
> yellow_group <- subset(group,favourite_colour=="yellow")
> mean(yellow_group$income)
[1] 2649.5
>  subset( group, favourite_colour %in% c("green","black") )$income
[1]  800 4000 1100 1900
> mean( group$income )
[1] 2047.429
> mean( subset(group,favourite_colour != "yellow")$income )
[1] 1806.6
> mean( subset(group,favourite_colour %in% c( "green","blue","black"))$income )
[1] 1806.6          # equivalent to previous command
```

You can split a data frame into a list of the respective subgroups. For example we split 'group' according to 'favourite_colour'. This is done with split() which returns a list of data frames. The command unsplit() reverses this and merges a list of data frames together into a single data frame.

```
> L <- split(group, group$favourite_colour)
> L            # L is a list of data frames
$black
   name gender favourite_colour income
4  Ines female            black   4000
7 Sarah female            black   1900

$blue
  name gender favourite_colour income
2 Caro female             blue   1233

$green
   name gender favourite_colour income
1  Hans   male            green    800
6 Peter   male            green   1100

$yellow
    name      gender favourite_colour income
3   Lars        male           yellow   2400
5 Samira      female           yellow   2899

> all( unsplit(L,group$favourite_colour)== group )  # identical?
[1] TRUE
```

If you wish to extend your data frame, then merge() might help you. Here is an example.

```
> snd_colour <- data.frame(favourite_colour=c("green","blue","yellow"),
+ second_colour=c("red","yellow","brown"))
> merge(group,snd_colour,all.x=TRUE)
  favourite_colour   name      gender income second_colour
1            black   Ines      female   4000          <NA>
2            black  Sarah      female   1900          <NA>
3             blue   Caro      female   1233        yellow
4            green   Hans        male    800           red
5            green  Peter        male   1100           red
6           yellow   Lars        male   2400         brown
7           yellow Samira      female   2899         brown
```

Note that everyone with `favourite_colour=="green"` has `second_colour=="red"`. The colour `"black"` is not associated with any colour, so the respective entries in the merged data frame remain `NA`. For more on merge(), see ?merge.

3.3 NA, Inf, NaN, NULL

Data sets are often not complete. There might be values which are simply not known. These missing values are recorded as NA (= not available). R deals quite well with missing data. Many commands have arguments to tell the command how to deal with NAs. The command for detecting missing values is is.na().

```
> v <- c( 1,3,NA,5 )
> v[1] <- NA
> v
[1] NA  3 NA  5
> is.na(v)
[1] TRUE FALSE  TRUE FALSE
> 5*v
[1]  NA 15 NA 25
> v*NA
[1] NA NA NA NA
> sum(v)
[1] NA
> exp(v)
[1]        NA  20.08554        NA 148.41316
> is.na(v)==FALSE
[1] FALSE  TRUE FALSE  TRUE
> w1 <- v[is.na(v)==FALSE]  # remove all missing values from the vector
> w1
[1] 3 5
> w2 <- v[!is.na(v)]        # same as w1
```

Many commands allow to ignore missing data. This is done with the argument 'na.rm=TRUE' (remove NA's) in commands which support this feature.

```
> sum(v,na.rm=TRUE)
[1] 8
```

The internal constants Inf and -Inf represent ∞ and $-\infty$. Everything outside a certain range is Inf or -Inf for R. This range depends on the system (32bit or 64bit machine). Another internal constant is NaN (not a number). Every calculation which is not defined results in NaN.

```
> 1.7e308
[1] 1.7e+308
> 1.8e308
[1] Inf
> exp(709)
[1] 8.218407e+307
> exp(710)
[1] Inf
> 5/0
[1] Inf
> exp(Inf)
[1] Inf
> Inf*(-2)
[1] -Inf
> Inf*Inf
[1] Inf
> 0/0
[1] NaN
> 0*Inf
[1] NaN
> Inf -Inf
[1] NaN
```

```
>1e-310
[1] 1e-310
>1e-330
[1] 0
```

'NULL' represents the null object (represents 'there is no object') in R. NULL is often returned by functions and expressions whose value is undefined. So NaN is an undefined numeric value and NULL is often used as undefined object.

3.4 Editing data

When invoked on a data frame or matrix, edit() opens a separate spreadsheet-like environment for editing. This is convenient for making small changes once a data set has been read. Note that edit(x) works on a local copy of the object, so without storing the result in a new object all changes are lost. We continue with the data frame 'group' from Subsection 3.2.

```
> edit(group)                # all changes will be lost
> newgroup <- edit(group)
> group <- edit(group)       # changes are stored back into group
> fix(group)                 # same as 'group <- edit(group)'
```

If edit() is invoked with a function, then an editor is opened which allows you to edit the definition of the function.

3.5 Reading and writing data frames

Read a data frame from a file with read.table(). Write a data frame to a file with write.table(). Typical call:

read.table("*filename.txt*",header=TRUE)
write.table(*dataframe*, file="*filename.txt*")

Before reading your data from a file into a data frame, you need to prepare the data file. Your data could look as follows:

weightcls	smoker	lifespan
3	0	50.5
3	0	52.8
3	1	54.7
3	0	56.0
2	0	58.1
2	1	60.2
2	0	62.8
2	0	64.5
1	1	66.3
1	0	68.4
1	0	70.2
1	1	72.1

Now start Excel and open a new table. Enter the data as in the above table. Then store the file as text file with the tabulator as delimitor: File/Save as/... then from the 'Save as type' options choose 'Text (Tab delimited)'. Store the file for example as 'lifespandata'. Excel automatically adds the extension '.txt' for text files. Alternatively open 'lifespandata.txt' with your favourite text editor and type in the above table. The field separator may be one or more spaces or tabs.

Make sure that the file is in the current working directory. Use the menu (File/change directory) or getwd() and setwd() to determine and set the working directory. Then you read the file with the command read.table(). The first line of the file contains the names of the variables and is called 'header', so we specify the option 'header=TRUE'.

```
> setwd("D:/Rcourse/")  # "" are necessary. Note / instead of \
> riscfactor <- read.table(file="lifespandata.txt",header=TRUE)
> # Alternatively use file.choose() (the choose-file-menu pops up in a GUI):
> # riscfactor <- read.table(file=file.choose(),header=TRUE)
> riscfactor
   weightcls smoker lifespan
1          3      0       50.5
2          3      0       52.8
3          3      1       54.7
4          3      0       56.0
5          2      0       58.1
6          2      1       60.2
7          2      0       62.8
8          2      0       64.5
9          1      1       66.3
10         1      0       68.4
11         1      0       70.2
12         1      1       72.1
> str(riscfactor)
'data.frame' : 12 obs. of  4 variables:
 $ weightcls : int  3 3 3 3 2 2 2 2 1 1 ...
 $ smoker    : int  0 0 1 0 0 1 0 0 1 0 ...
 $ lifespan  : num  50.3 52.8 54.7 56 58.1 60.2 62.8 64.5 66.3 68.4 \ldots
> write.table(riscfactor,file="lifespanout.txt")
```

It is important to learn how to put your data into the data file or Excel spreadsheet. There are countless ways of doing it but only one way which makes life easy later. The key thing to remember is that

all the values of the same variable go in the SAME COLUMN.

Here is an example. If you had an experiment with three treatments (control, pre-heated, pre-chilled), and four measurements per treatment, it might seem like a good idea to create the spreadsheet like this:

Control	Pre-heated	Pre-chilled
6.1	6.3	7.1
5.9	6.2	8.2
5.8	5.8	7.3
5.4	6.3	6.9

However this is not ideal for handling the data with R. The good way to enter these data is to have one column for the response variable and one column which indicates the treatment.

Response	Treatment
6.1	Control
5.9	Control
5.8	Control
5.4	Control
6.3	Pre-heated
6.2	Pre-heated
5.8	Pre-heated
6.3	Pre-heated
7.1	Pre-chilled
8.2	Pre-chilled
7.3	Pre-chilled
6.9	Pre-chilled

This organization of the data is more suitable for R as R is particularly good in grouping a vector (here 'Response') according to a criterium (here 'Treatment'). Later we will learn 'Response ˜ Treatment' for this. It is more work in R to join vectors together.
The following two lists explain important options of `read.table()` and of `write.table()` in more detail.

read.table(file, header = FALSE, sep = "", dec = ".", row.names, fill = FALSE, ...)

header: a logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: 'header' is set to 'TRUE' if and only if the first row contains one fewer field than the number of columns.

sep: the field separator character. Values on each line of the file are separated by this character. If 'sep = ""' (the default for 'read.table') the separator is 'white space', that is one or more spaces, tabs, newlines or carriage returns.

row.names: a vector of row names. This can be a vector giving the actual row names, or a single number giving the column of the table which contains the row names, or character string giving the name of the table column containing the row names.

If there is an header and the first row contains one fewer field than the number of columns, the first column in the input is used for the row names. Otherwise if 'row.names' is missing, the rows are numbered.

fill: logical. If 'TRUE' then in case the rows have unequal length, blank fields are implicitly added. Otherwise exit with an error message.

... There are more options, see ?read.table.

write.table(x,file="", append=FALSE, quote=TRUE, sep=" ", eol="\n", dec = ".",
row.names=TRUE,...)

x: the object to be written, preferably a matrix or data frame. If not, it is attempted to coerce 'x' to a data frame.

file: a character string naming a file for writing. '""' indicates output to the console.

append: logical. If 'TRUE', the output is appended to the file. If 'FALSE', any existing file of the name is destroyed. So be careful.

quote: a logical value ('TRUE' or 'FALSE') or a numeric vector. If 'TRUE', any character or factor columns will be surrounded by double quotes. If a numeric vector, its elements are taken as the indices of columns to quote. In both cases, row and column names are quoted if they are written. If 'FALSE', nothing is quoted.

sep: the field separator string. Values within each row of 'x' are separated by this string.

dec: the string to use for decimal points in numeric or complex columns: must be a single character.

row.names: either a logical value indicating whether the row names of 'x' are to be written along with 'x', or a character vector of row names to be written.

... There are more options, see ?write.table.

There are short-cuts for different types of input files which save typing. Here is a list which specifies the defaults for the different commands.

Command	header	sep	dec	fill
read.table()	FALSE	""	"."	FALSE
read.csv()	TRUE	","	"."	TRUE
read.csv2()	TRUE	";"	","	TRUE
read.delim()	TRUE	"\t"	"."	TRUE
read.delim2()	TRUE	"\t"	","	TRUE

The commands `read.csv2()` and `read.delim2()` are for input files with german decimal notation. Note that 'csv' is short for 'comma-separated values'. The analogs of `write.table()` are `write.csv()`, `write.csv2()`.

3.6 Examples of different input files

Before reading in an input file, you should open the data file in order to find out its format. The following examples show the first lines of the input file and specify commands for reading the data.

Example 1: First three lines of the file are:

```
wghtcls "smoker" lifespan
"3" 0 50.3
3   0 52.8
```

R command (string "3" is converted to 3)

```
> riscfactor <- read.table("lifespandata2.txt",header=TRUE)
```

Example 2: First three lines of the file are:

```
wghtcls,smoker,lifespan
3,0,50.3
3,0,52.8
```

Two R commands doing the same:

```
> riscfactor <- read.csv("lifespandata.csv")
> riscfactor <- read.table("lifespandata.csv",header=TRUE,sep=",",fill=TRUE)
```

Example 3: First three lines of the file are:

```
wghtcls;smoker;lifespan
3;0;50,3
3;0;52,8
```

R command

```
> riscfactor <- read.csv2("lifespandata.csv2")
```

Example 4: First three lines of the file are:

```
weight class smoker lifespan
3 0 50.3
3 0 52.8
```

Note that the name of the first variable contains a space. This is a problem if spaces are also delimiters. Here R expects 4 entries per line. The R command

```
> riscfactor <- read.table("lifespandataspace.txt",header=TRUE)
```

therefore results in an error. Adding the option `fill=TRUE` results in 4 variables with the fourth variable `lifespan` having only `NA` entries. You avoid this problem, if you replace 'weight class' e.g. by 'weight.class' or 'weightclass' or '"weight class"' or if you replace the separater by a non-white-space such as in the file 'lifespandataspace.csv'.

3.7 Factors

In the data frame 'riscfactors', the values of 'weightcls' are of type 'numeric'. However, the values '1', '2' and '3' in the vector 'weightcls' are intended to be names of different groups rather than true numerical values. This does make a difference to R so we need to tell R what our intention is. This is done with factor(). The command levels() returns the names of the different groups in a factor. Note the different behaviour of str() and of summary() according to whether the argument is a numeric vector or a factor.

```
> x <- c("female","male","male","female","female")
> levels(x)
NULL
> str(x)
 chr [1:5] "female" "male" "male" "female" "female"
> x <-factor(x)
> levels(x)
[1] "female" "male"
> str(x)
 Factor w/ 2 levels "female","male": 1 2 2 1 1
> y <- rep(c(17,17,18),4); str(y)
 num [1:12] 17 17 18 17 17 18 17 17 18 17 ...
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  17.00   17.00   17.00   17.33   18.00   18.00
> y <- factor(y); str(y)
```

```
 Factor w/ 2 levels "17","18": 1 1 2 1 1 2 1 1 2 1 ...
> summary(y)
17 18
 8  4
```

So if numerical values are used as names rather then as true numerical values, then mark the vector as factor. The following definition of riscfactors2 improves the definition of riscfactors.

```
> riscfactors2 <- data.frame( weightcls=factor( rep(3:1,c(4,4,4)) ),
+ smoker=factor( rep(c(0,0,1),4) ), lifespan=seq(50,72,2) )
> summary(riscfactors2)
   weightcls smoker    lifespan
 1 1:4       0:8    Min.   :50.0
 2 2:4       1:4    1st Qu.:55.5
 3 3:4              Median :61.0
 4                  Mean   :61.0
 5                  3rd Qu.:66.5
 6                  Max.   :72.0
 (Other):6
> summary(riscfactors)
   weightcls     smoker          lifespan
 Min.   :1   Min.   :0.0000   Min.   :50.0
 1st Qu.:1   1st Qu.:0.0000   1st Qu.:55.5
 Median :2   Median :0.0000   Median :61.0
 Mean   :2   Mean   :0.3333   Mean   :61.0
 3rd Qu.:3   3rd Qu.:1.0000   3rd Qu.:66.5
 Max.   :3   Max.   :1.0000   Max.   :72.0
```

If the column of a data file is a factor, then either specify the class of the columns with `colClasses=` or convert the variable after reading the data:

```
> riscfactor <- read.table("lifespandata.txt",header=TRUE,
+       colClasses=c("factor","numeric","numeric"))
> riscfactor <- read.table("lifespandata.txt",header=TRUE)
> class(riscfactor$wghtcls)
[1] "integer"
> riscfactor$wghtcls <- factor(riscfactor$wghtcls)
> class(riscfactor$wghtcls)
[1] "factor"
```

4 Plotting

There are three types of plotting commands:

- **High-level** plotting functions create a new plot (usually with axes, labels, titles and so on).
- **Low-level** plotting functions add more information to an existing plot, such as extra points, lines or labels.
- **Interactive** graphics functions allow you to interactively add information to an existing plot or to extract information from an existing plot using the mouse.

4.1 High-level plotting commands

The standard high-level plotting function is plot(). The beaviour of this command depends on the type of its argument. Here is a selection of possible arguments and the resulting plot.

plot(x, y) If x and y are numerical vectors, then plot(x, y) produces a scatterplot of y against x.

plot(y) If y is a numerical vector, then this is (almost) the same as plot($1 : \text{length}(y), y$).

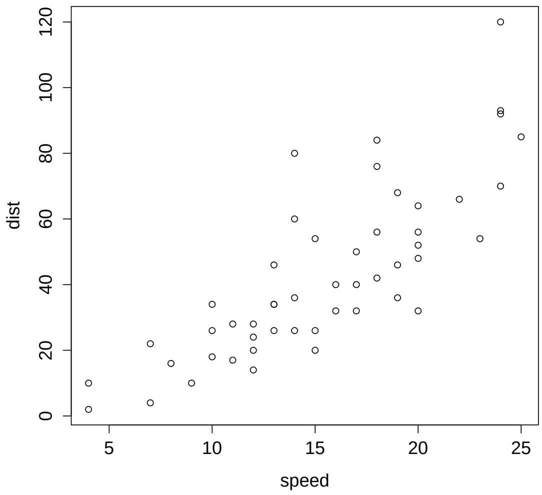
plot(f) If f is a factor, then plot(f) is a barplot of f.

plot(f, y) If f is a factor and y is a numeric vector, then plot(f, y) produces boxplots of y for each level of f.

plot(*fun*) If *fun* is a function, then plot(*fun*, from=a, to=b) plots *fun* in the range $[a, b]$.

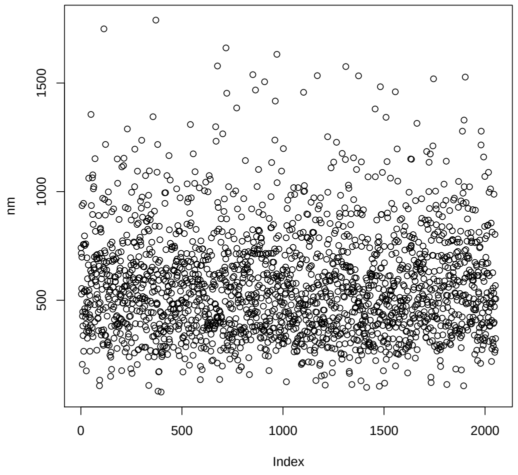
Distance needed to stop (in ft) from a certain speed (mph):

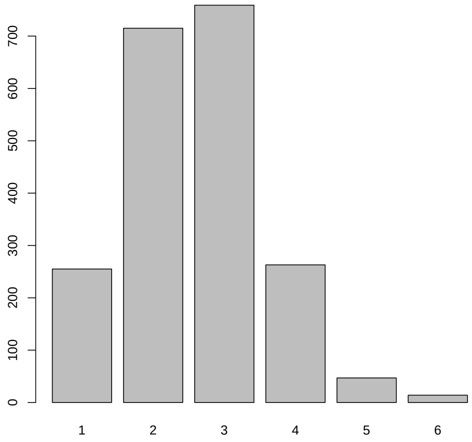
```
> data(cars)    # cars is a dataset in the library 'datasets', see ?cars
> attach(cars)
> str(cars)
> ?cars
> plot(speed,dist)
```



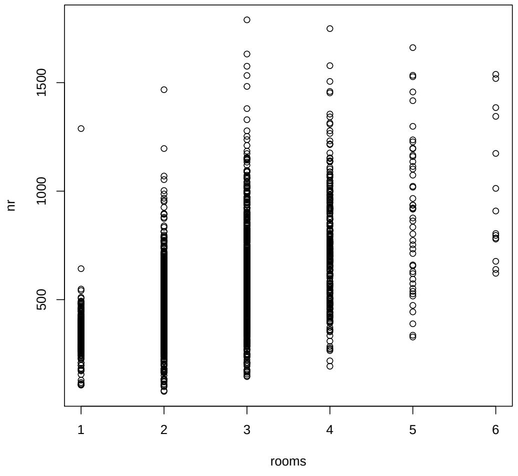
The dataset for the following examples of `plot()` is described in the file *miete03.readme.txt* which can be downloaded from the homepage.

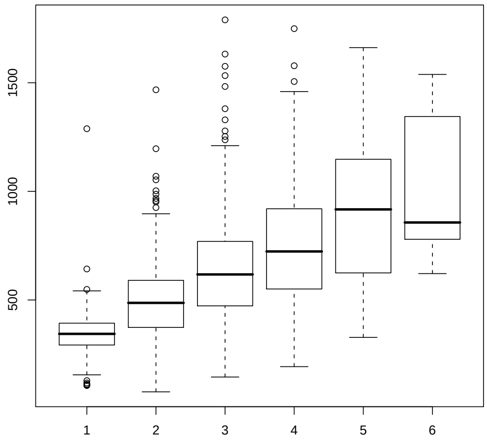
```
> rent <- read.table("miete03.asc",header=TRUE)
> attach(rent)
> str(rent)
'data.frame': 2053 obs. of  13 variables:
 $ nm      : num  741 716 528 554 698 ...
 $ nmqm    : num  10.9 11.01 8.38 8.52 6.98 ...
 $ wfl     : int  68 65 63 65 100 81 55 79 52 77 ...
 $ rooms   : int  2 2 3 3 4 4 2 3 1 3 ...
 $ bj      : num  1918 1995 1918 1983 1995 ...
 $ bez     : int  2 2 2 16 16 16 6 6 6 6 ...
 $ wohngut : int  1 1 1 0 1 0 0 0 0 0 ...
 $ wohnbest: int  0 0 0 0 0 0 0 0 0 0 ...
 $ ww0     : int  0 0 0 0 0 0 0 0 0 0 ...
 $ zh0     : int  0 0 0 0 0 0 0 0 0 0 ...
 $ badkach0: int  0 0 0 0 0 0 0 0 0 0 ...
 $ badextra: int  0 0 0 1 1 0 1 0 0 0 ...
 $ kueche  : int  0 0 0 0 1 0 0 0 0 0 ...
> plot(nm)                # 'nm' is a numerical vector
> plot(factor(rooms))     # 'rooms' is converted into a factor
```



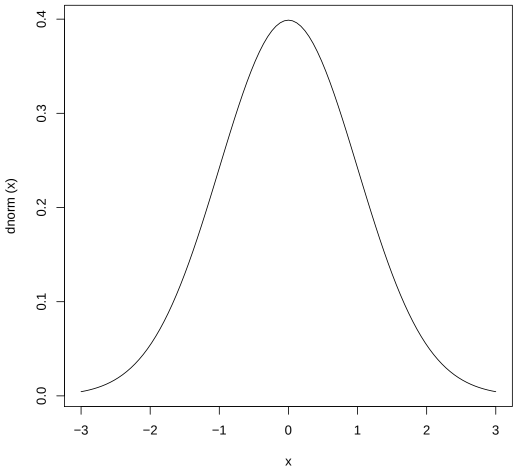


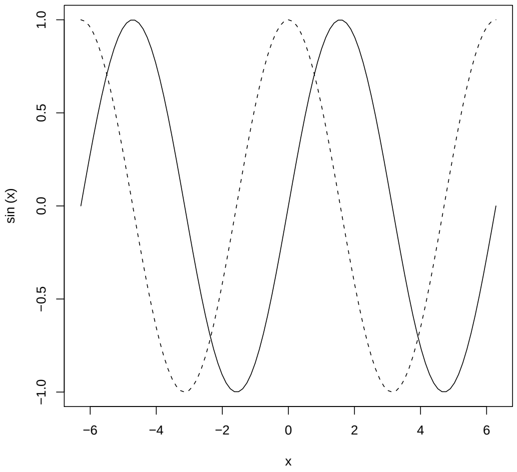
```
> plot(rooms,nm)          # Here rooms is a numerical vector
> plot(factor(rooms),nm) # Here rooms is converted into a factor
```





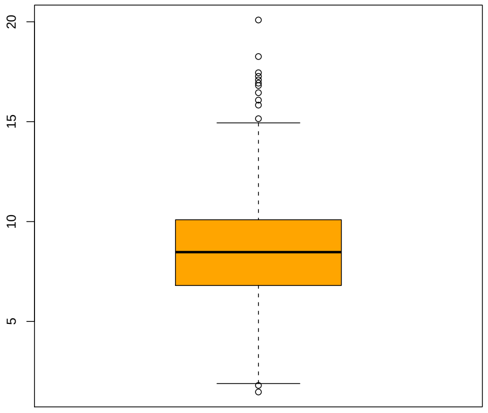
```
> plot(dnorm,from=-3,to=3)
> plot(sin,from=-2*pi,to=2*pi)
> plot(cos,from=-2*pi,to=2*pi,add=TRUE,lty="dashed")
```

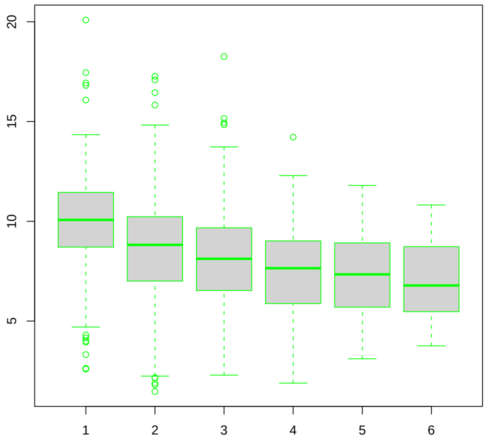




Next we boxplot the net rent per m^2. Then we wish to know whether there is a connection between the net rent per m^2 and the number of rooms in the flat. For this we wish to split the vector 'nmqm' according the factor 'rooms'. This is done with the ˜-operator. The expression 'nmqm˜ rooms' groups 'nmqm' according to the levels in 'rooms'

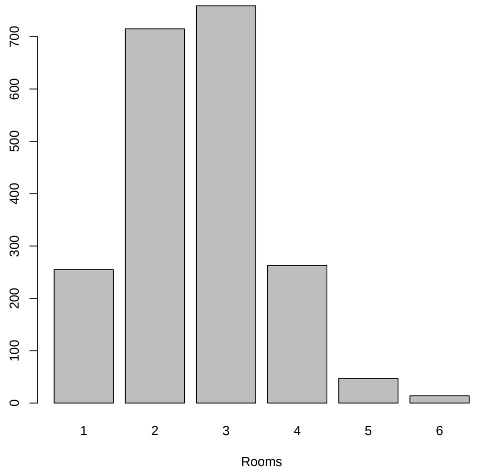
```
> boxplot(nmqm,col='orange')
> boxplot(nmqm~rooms,col='lightgray',border='green')
# The last command is the same as
> boxplot(list( nmqm[which(rooms==1)], nmqm[which(rooms==2)],
+  nmqm[which(rooms==3)], nmqm[which(rooms==4)], nmqm[which(rooms==5)],
+  nmqm[which(rooms==6)] ),col='lightgray',border='green')
```

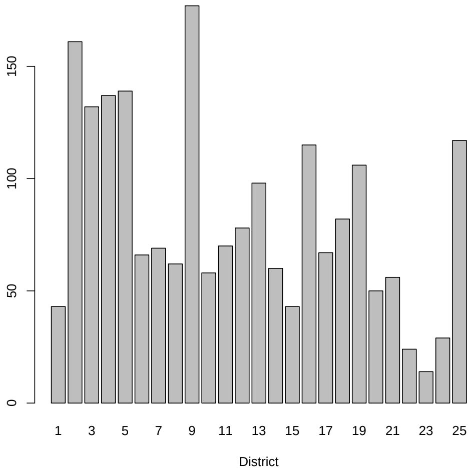




Now we visualize 'rooms' with barplot(). For this we need to create a table which gives the number of flats with 1 room, the number of flats with 2 rooms and so on. This is done by table().

```
> barplot(table(rooms),xlab="Rooms")
> barplot(table(bez),xlab="District")
```





There are a number of arguments which may be passed to high-level graphics functions. The following list contains a selection of the most important arguments.

`add=TRUE` Forces the function to act as a low-level graphics function, superimposing the plot on the current plot (does not work reliably).

`type=` The `type=` argument controls the type of plot produced, as follows:

- `type="p"` Plot individual points (the default)
- `type="l"` Plot lines
- `type="b"` Plot points connected by lines (both)
- `type="o"` Plot points overlaid by lines
- `type="h"` Plot vertical lines from points to the zero axis (high-density)
- `type="s"`
- `type="S"` Step-function plots. In the first form, the top of the vertical defines the point; in the second, the bottom.
- `type="n"` No plotting at all. However axes are still drawn (by default) and the coordinate system is set up according to the data. Ideal for creating plots with subsequent low-level graphics functions.

`xlab=string`

`ylab=string` Axis labels for the x and y axes. Use these arguments to change the default labels, usually the names of the objects used in the call to the high-level plotting function.

`main=string` Figure title, placed at the top of the plot in a large font.

`sub=string` Sub-title, placed just below the x-axis in a smaller font.

`axes=FALSE` Suppresses generation of axes. Useful for adding your own custom axes with the axis() function. The default `axes=TRUE` includes axes.

`log="x"`

`log="y"`

`log="xy"` Causes the x, y or both axes to be logarithmic.

`cex=1` Amount by which plotting text and symbols should be magnified relative to the default. This option is used if the default size of text is too small.

Graphic parameters can also be changed permanently with the command par(). For example

```
> par(col="green",lty="dashed")
```

sets the colour permanently to "green" and the line type to "dashed". To undo this later proceed as follows.

```
> oldpar <- par(col="green",lty="dashed") # store oldsetting in 'oldpar'
> # ... plotting commands ...
> par(oldpar)                             # resets all parameters
```

4.2 Low-level plotting functions

Low-level plotting commands can be used to add extra information (such as points, lines or text) to the current plot. Some of the more useful low-level plotting functions are:

`points(`x`,` y`)`

`lines(`x`,` y`)` Adds points or connected lines to the current plot.

`text(`x`,` y`,` *`labels`*`, ...)` Add text to a plot at points given by x, y. Normally `labels` is an integer or character vector in which case `labels[i]` is plotted at point `(x[i], y[i])`.

`abline(`a, b`)`

`abline(h=`y` )`

`abline(v=`x` )`

`abline(`*`lm.obj`*` )` Adds a line of slope b and intercept a to the current plot. h=y may be used to specify y-coordinates for the heights of horizontal lines to go across a plot, and v=x similarly for the x-coordinates for vertical lines. Also *lm.obj* may be a list with a coefficients component of length 2 (such as the result of model-fitting functions,) which are taken as an intercept and slope, in that order.

`title(main=`*`main`*`,sub=`*`sub`*`)` Adds a title *main* to the top of the current plot in a large font and (optionally) a sub-title *sub* at the bottom in a smaller font.

`axis(side=`*`side`*`, ...)` Adds an axis to the current plot on the side given by the first argument (1 to 4, counting clockwise from the bottom.) Other arguments control the positioning of the axis within or beside the plot, and tick positions and labels. Useful for adding custom axes after calling plot() with the `axes=FALSE` argument.

`legend(`x`,` y`, legend, ...)` Adds a legend to the current plot at the specified position. Plotting characters, line styles, colors etc., are identified with the labels in the character vector legend. At least one other argument v (a vector the same length as legend) with the corresponding values of the plotting unit must also be given, as follows:

`legend( , fill=`v` )` Colors for filled boxes

`legend( , col=`v` )` Colors in which points or lines will be drawn

`legend( , lty=`v` )` Line styles

`legend( , lwd=`v` )` Line widths

`legend( , pch=`v` )` Plotting characters (character vector)

`polygon(`x`, `y`, ...)` Draws a polygon defined by the ordered vertices in (x, y) and (optionally) shade it in with hatch lines, or fill it if the graphics device allows the filling of figures.

`arrows(`$x0$`,`$y0$`,`$x1$`,`$y1$`, ...)` Draw arrows from $(x0, y0)$ to $(x1, y1)$.

`p.arrows(`$x0$`,`$y0$`,`$x1$`,`$y1$`, ...)` Same as `arrows()` but with colour filled arrow heads. Requires library 'sfsmisc'.

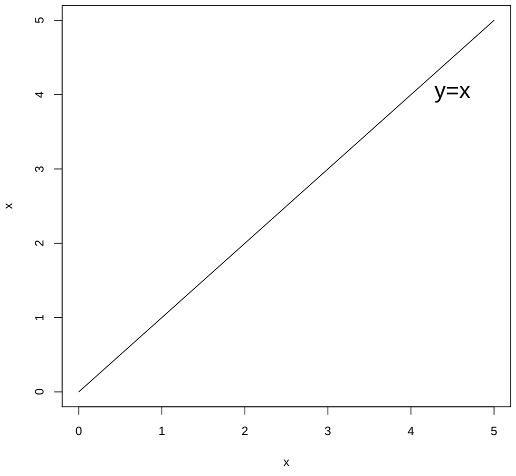
`pretty()` Calculate a 'pretty' scaling of the axis.

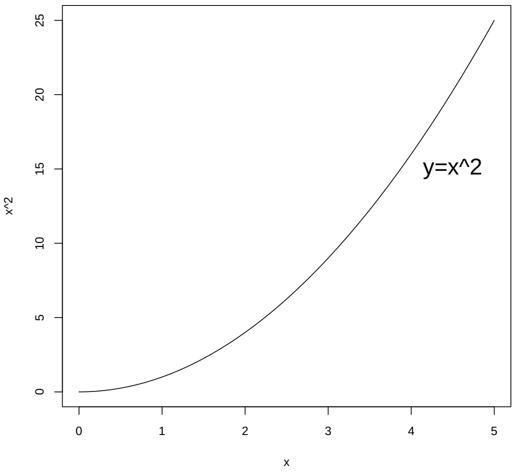
`plot.new()` Empty the current plotting window (open a new window if none is open).

`mtext()` Write text in the margins.

Next we introduce a feature to add mathematical symbols to a plot. The following example plots the function $y = x$ and uses a character string to label it.

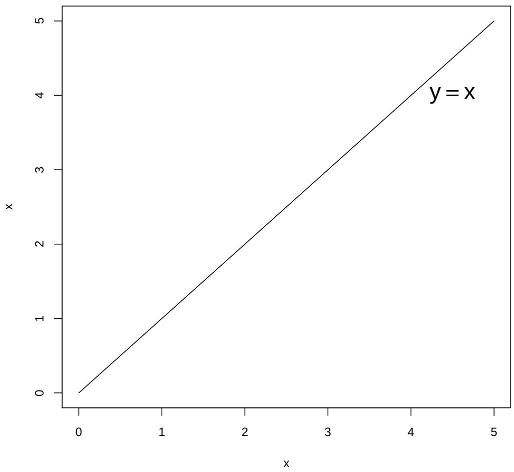
```
> x <- seq(from=0,to=5,by=0.1)
> plot(x,x,type="l")
> text(4.5,4,"y=x",cex=2)
> plot(x,x^2,type="l")
> text(4.5,15,"y=x^2",cex=2)
```

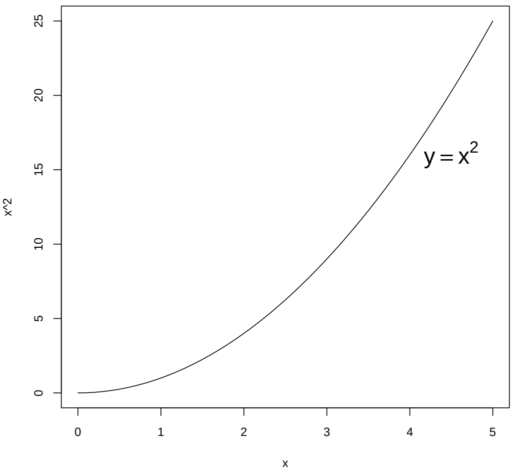




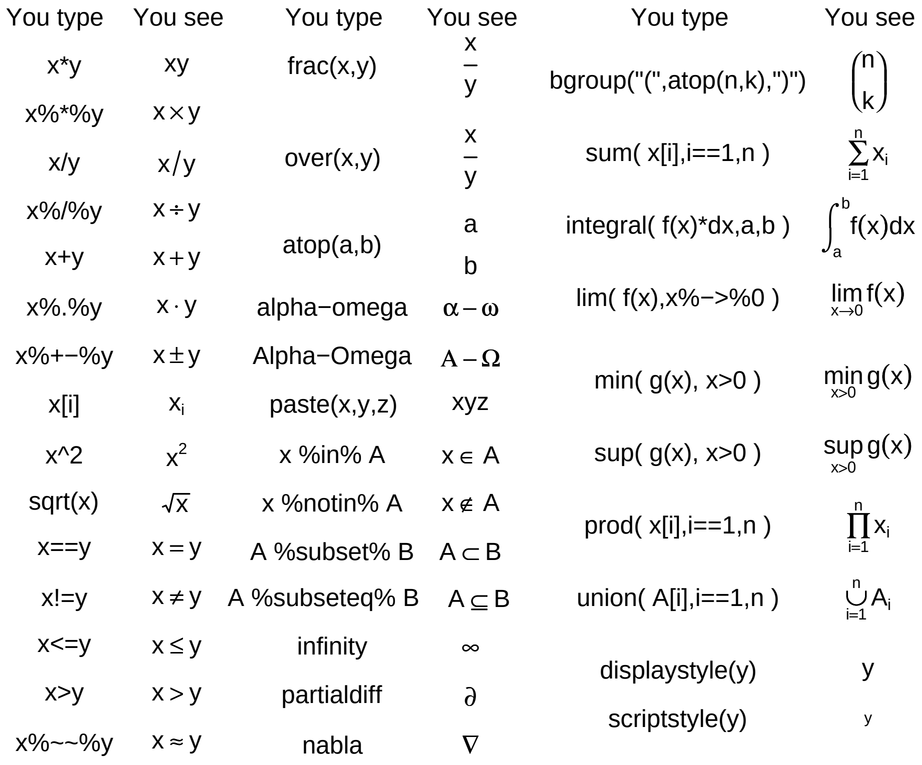
As you can see from the last example with `y=x^2`, realizing mathematical symbols with plain text is not a good solution. R provides a solution which uses so-called 'expressions'. If the 'text' argument to one of the text-drawing functions ('text', 'mtext', 'axis', 'legend') in R is an expression, then the argument is interpreted as a mathematical expression and the output will be formatted according to TeX-like rules. An expression is created with the command `expression()`. Here are some examples

```
> x <- seq(from=0,to=5,by=0.1)
> plot(x,x,type="l")
> text(4.5,4,expression(y==x),cex=2)
> plot(x,x^2,type="l")
> text(4.5,15,expression(y==x^2),cex=2)
```





Here is a selection of mathematical notations which can be used to print mathematical notation into plots.



You type	You see	You type	You see	You type	You see
x*y	xy	frac(x,y)	$\frac{x}{y}$	bgroup("(",atop(n,k),")")	$\binom{n}{k}$
x%*%y	$x \times y$				
x/y	x/y	over(x,y)	$\frac{x}{y}$	sum(x[i],i==1,n)	$\sum_{i=1}^{n} x_i$
x%/%y	$x \div y$	atop(a,b)	$\genfrac{}{}{0pt}{}{a}{b}$	integral(f(x)*dx,a,b)	$\int_a^b f(x)dx$
x+y	$x + y$				
x%.%y	$x \cdot y$	alpha−omega	$\alpha - \omega$	lim(f(x),x%−>%0)	$\lim_{x \to 0} f(x)$
x%+−%y	$x \pm y$	Alpha−Omega	$A - \Omega$	min(g(x), x>0)	$\min_{x>0} g(x)$
x[i]	x_i	paste(x,y,z)	xyz		
x^2	x^2	x %in% A	$x \in A$	sup(g(x), x>0)	$\sup_{x>0} g(x)$
sqrt(x)	$\sqrt{x}$	x %notin% A	$x \notin A$	prod(x[i],i==1,n)	$\prod_{i=1}^{n} x_i$
x==y	$x = y$	A %subset% B	$A \subset B$		
x!=y	$x \neq y$	A %subseteq% B	$A \subseteq B$	union(A[i],i==1,n)	$\bigcup_{i=1}^{n} A_i$
x<=y	$x \leq y$	infinity	∞	displaystyle(y)	y
x>y	$x > y$	partialdiff	∂	scriptstyle(y)	$\scriptstyle y$
x%~~%y	$x \approx y$	nabla	∇		

More information, including a full listing of the features available can be obtained from within R using the commands:

```
> demo(plotmath)
> help(plotmath)
> example(plotmath)
```

If you need special symbols, then Hershey characters might help. For an overview, type

```
> demo(Hershey)
> help(Hershey)
> example(Hershey)
```

4.3 Interacting with plots

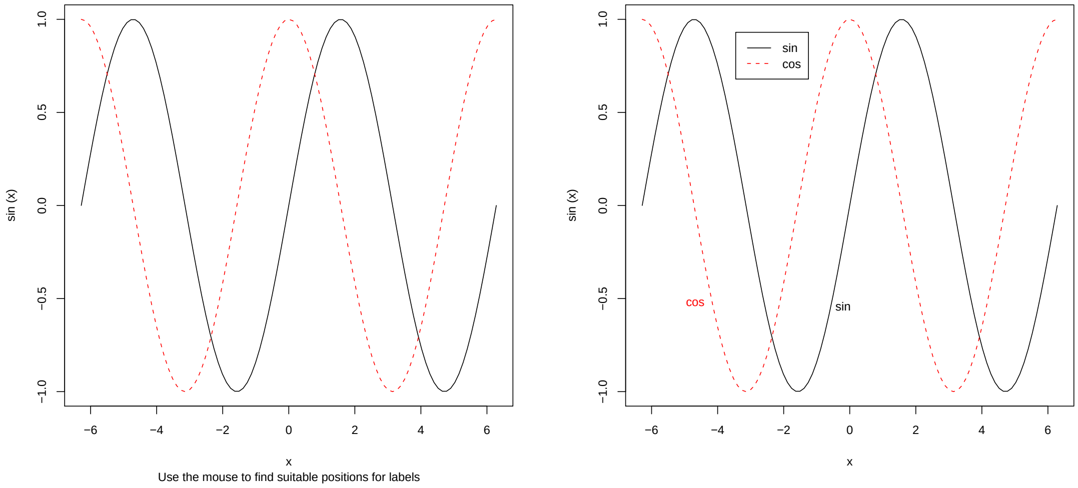
R provides functions which allow users to extract or add information to a plot using a mouse.

`locator(`*n*`,type="n")` Waits for the user to select locations on the current plot using the left mouse button. This continues until n (default 512) points have been selected, or another mouse button is pressed. The type argument allows for plotting at the selected points and has the same effect as for high-level graphics commands; the default is no plotting. locator() returns the locations of the points selected as a list with two components x and y.

identify(*x*,*y*,*labels*) Allow the user to highlight any of the points defined by x and y (using the left mouse button) by plotting the corresponding component of *labels* nearby (or the index number of the point if labels is absent). Returns the indices of the selected points when another button is pressed.

For example you could use these to identify outliers in your data or to find an appropriate position for the legend in your plot.

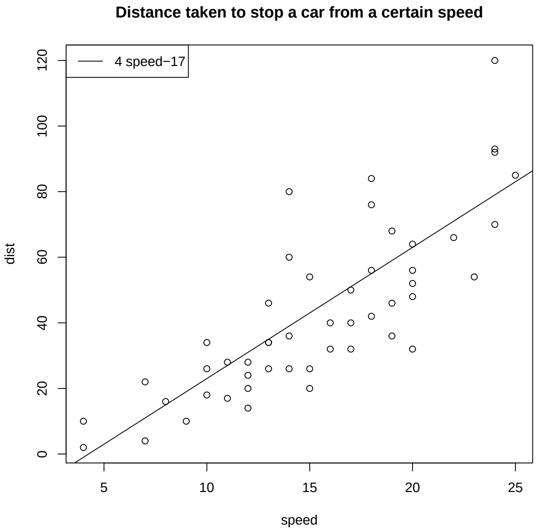
```
> plot(sin,from=-2*pi,to=2*pi,)
> plot(cos,from=-2*pi,to=2*pi,add=TRUE,lty="dashed",col="red")
> text(locator(1),"sin",adj=0)
> text(locator(1),"cos",adj=1,col="red")
> legend(locator(1),legend=c("sin","cos"),col=c("black","red"),lty=c(1,2))
```



4.4 Plotting examples

Example 1: Cars

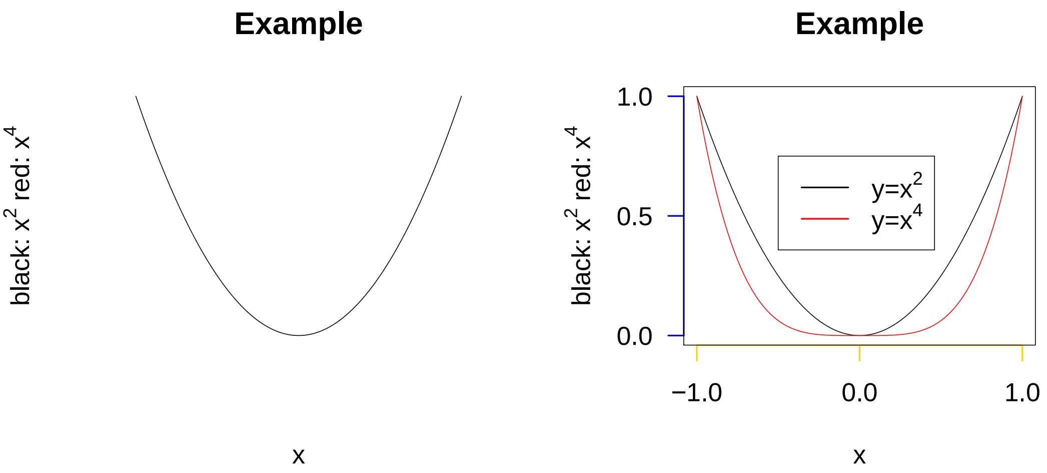
```
> attach(cars)
> str(cars)
> plot(speed,dist)
> legend("topleft",legend="4 speed-17",lty="solid")
> abline(-17,4)
> title(main="Distance taken to stop a car from a certain speed")
```



Distance taken to stop a car from a certain speed

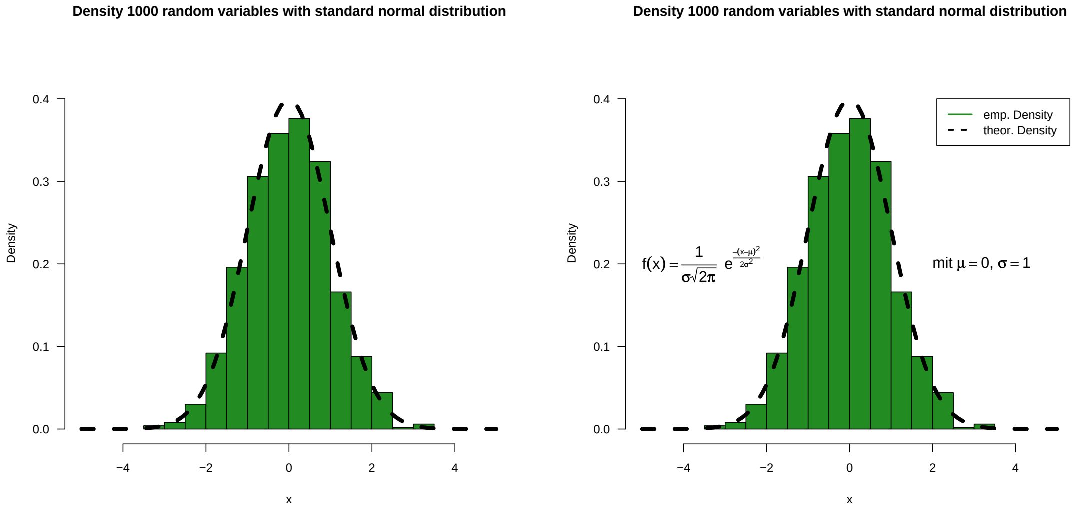
Example 2:

```
> oldpar <- par(las=1,cex=2)
> x <- seq(-1,1,by=0.01)
> y1 <- x^2
> y2 <- x^4
> plot(x,y1, type="l",axes=FALSE,main="Example",xlab="x",
+  ylab=expression(paste("black: ", x^2, " red: ", x^4)) )
>
> lines(x,y2, type="l", col=2)
> axis(1, at=c(-1, 0, 1), labels=c("-1.0", "0.0", "1.0" ),col = "gold",
+  lty = "solid", lwd = 2 )
> axis(2, at=c(0, 0.5, 1.0), labels=c("0.0", "0.5", "1.0" ),col = "blue",
+  lty = "solid", lwd = 2 )
> legend( -0.5,0.75,legend=c(expression(y==x^2), expression(y==x^4)),
+         col=c("black", "red"), lwd=2 )
> box()
> par(oldpar)
```



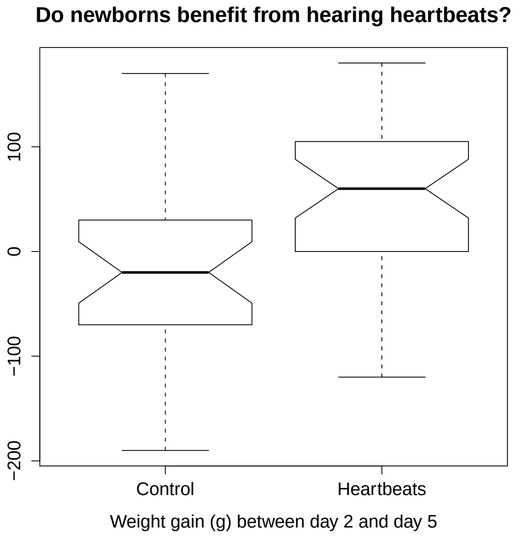
Example 3: Empirical and theoretical density of the standard normal distribution.

```
> par(las=1)      # all labellings of axes are horizontal
> hist(x, main="Density 1000 random variables with standard normal
+   distribution", probability=TRUE, col="forestgreen", ylab="Density",
+   xlim=c(-5,5),ylim=c(0,0.45))
> plot(dnorm,from=-5,to=5,add=TRUE,lwd=5,lty="dashed")
> text(-5,0.2, adj=0, cex=1.3, expression(f(x)==frac(1,sigma*sqrt(2*pi))~~
+   e^{frac(-(x-mu)^2, 2*sigma^2)}))
> text(2,0.2, adj=0, cex=1.3, expression("mit "*{mu==0}*", "*{sigma==1}))
> legend(2.1,0.4,legend=c("emp. Density","theor. Density"),
+   col=c("forestgreen","black"), lwd=2,lty=c(1,2))
```



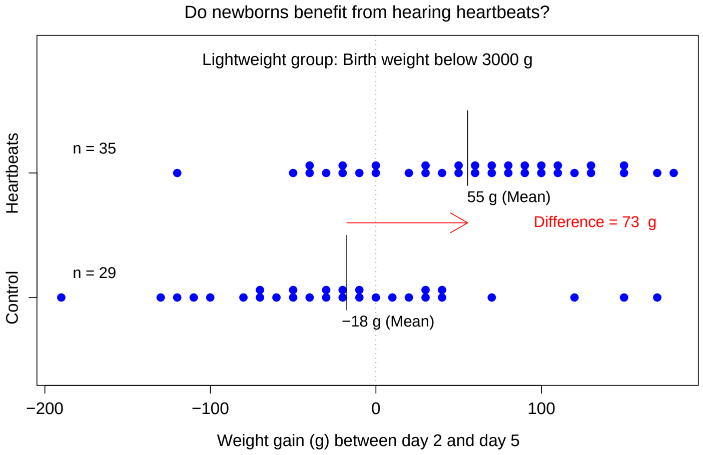
Example 4: Do newborns benefit from hearing heartbeats?

```
> data<-read.table("heartbeats.txt",header=TRUE)
> attach(data)
> str(data)
'data.frame': 210 obs. of  3 variables:
$ wghtcls  : int  1 1 1 1 1 1 1 1 1 1 ...
$ treatment: int  0 0 0 0 0 0 0 0 0 0 ...
$ wghtincr : int  -190 -130 -120 -110 -100 -80 -70 -70 -60 -50 ...
> h<-treatment[wghtcls==1]
> i<-wghtincr[wghtcls==1]
> boxplot(i~h,names=c("Control","Heartbeats"),
+         xlab="Weight gain (g) between day 2 and day 5",notch=TRUE)
> title(main="Do newborns benefit from hearing heartbeats?")
> detach(data)
```



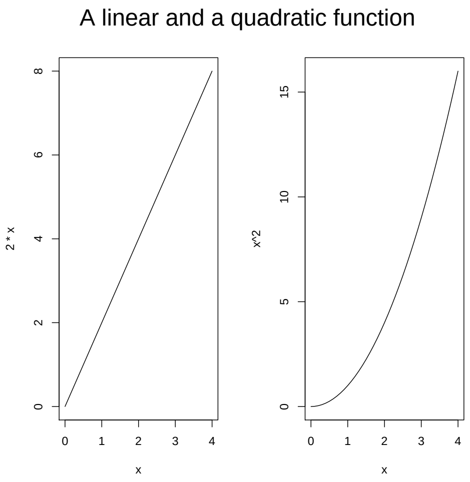
Example 5: Do newborns benefit from hearing heartbeats?

```
> data<-read.table("heartbeats.txt",header=TRUE)
> attach(data)
> h<-treatment[wghtcls==1]
> i<-wghtincr[wghtcls==1]
> mtitle<-"Do newborns benefit from hearing heartbeats?"
> sbtitle<-"Lightweight group: Birth weight below 3000 g"
> xlb<-"Weight gain (g) between day 2 and day 5"
> dev.new(height=6,width=9.5)
> par(cex.main=1.5,cex.axis=1.4,cex.lab=1.4,font.main=1,mar=c(5,5,3,1))
> stripchart(i~h,method="stack",col="blue",pch=16,cex=1.4,
+   group.names=c("Control","Heartbeats"),ylim=c(0.4,3),main=mtitle,xlab=xlb)
> text((min(i)+max(i))/2,2.9,sbtitle,cex=1.4)
> abline(v=0,lty=3)
> m0<-mean(i[h==0])
> m1<-mean(i[h==1])
> lines(c(m0,m0),c(0.9,1.5))
> text(m0+25,0.8,paste(round(m0),"g (Mean)"),cex=1.3)
> lines(c(m1,m1),c(0.9+1,1.5+1))
> text(m1+25,0.8+1,paste(round(m1),"g (Mean)"),cex=1.3)
> arrows(m0,1.6,m1,1.6,col="red")
> text(m1+40,1.6,paste("Difference =",round(m1-m0)," g"),cex=1.3,col="red",
+   adj=0)
> n0<-length(i[h==0])
> n1<-length(i[h==1])
> text(-170,1.2,paste("n =",n0),cex=1.3)
> text(-170,1.2+1,paste("n =",n1),cex=1.3)
> detach(data)
```



Example 6: A multiplot.

```
> op<-par(mfrow=c(1,2))    # save old plotting options in 'op'
> x <- seq(0,2,by=0.1)
> plot(x,2*x,type="l")
> plot(x,x^2,type="l")
> mtext("A linear and a quadratic function",outer=TRUE,side=3,line=-2,cex=2)
> par(op)                  # reset plotting options
```



4.5 Devices

Having produced a nice plot, the next step is often to store the plot in a pdf-file. Here is how to do this.

```
> plot(...)  # Begin a plot with an high-level plotting function such as plot()
> ...        # Further low-level plotting function enrich the plot
# After you are finished with the plot:
> dev.print(device=pdf, file="filename.pdf" )
```

Now the file `filename.pdf` contains the same plot which you saw on the screen. The rest of the subsection explains further means of plotting into files.

Plots can be printed into windows and into files. The word 'device' is used to refer both to windows and to files. The command dev.new() opens a window which becomes the active device. At any time there is exactly one active device (or no device at all). All graphical operations occur on the active device. The command plot.new() is used to delete all contents of the active device (starts a new plot). If there is no active device, then plot.new() opens a window which becomes the active device. All high-level plotting functions first of all call plot.new(). So if you plot two graphs with plot(), then the second graph overwrites the first graph. If you wish to view both graphs simultaneously, then call dev.new() before executing the second plot() command. There is a list of open devices, and this is considered as a circular numbered list. The device with number 1 is always the 'null device' which is really a placeholder; any attempt to use it will open a new device. The following list of commands enables you to handle this device list.

`dev.new(height=7,width=7)` Opens a new window which then becomes the active device. The default size is a 7 inches square.

`dev.off()` Closes the active device. The next device in the device list then becomes active.

`graphics.off()` Closes all open devices.

`plot.new()` Deletes all contents of the active device. If no device is open, then plot.new() opens a new window which becomes the active device.

`dev.set(`*n*`)` Make the device with number n the active device.

`dev.prev()`
`dev.cur()`
`dev.next()` Return the number and name of the previous/current/next device in the list of devices.

`dev.list()` List all open devices.

`dev.print(device=`*dev*`,file="`*filename*`")` Copy the content of the active device to the file *"filename"*. The type of output is specified by the device *dev* which can be e.g. pdf, postscript, jpeg, bitmap, pictex, xfig, bmp, png.

`dev.copy(device=`*dev*`,file="`*filename*`")` Same as dev.print() but does not close the device.

`dev.print(which=`*n*`)` Copy the content of the active device to the device with number n

`dev.copy2pdf(file="`*filename.pdf*`")` Same as dev.copy(device=pdf,file=*"filename.pdf"*).

`dev.copy2eps(file="`*filename.eps*`")` Copies contents of the active device into an eps-file.

An alternative to copying a device to a file is to plot directly into a file. This is useful when writing R scripts. For example, pdf(”filename.pdf”) opens the pdf-file *filename.pdf* as pdf-device. All graphical operations occur then directly on this pdf-device. Note that the pdf-file will often only be created when the pdf-device is closed with `dev.off()`. Here is a list of commands which open devices.

`pdf("`*`filename`*`")` Opens the pdf-file *filename* as device.

`postscript("`*`filename`*`")` Opens the postscript-file *filename* as device.

`jpeg("`*`filename`*`")` Opens the jpeg-file *filename* as device.

`bitmap("`*`filename`*`")` Opens the bitmap-file *filename* as device.

`tiff("`*`filename`*`")` Opens the tiff-file *filename* as device.

`xfig("`*`filename`*`")` Opens the xfig-file *filename* as device.

`pictex("`*`filename`*`")` Opens the pictex-file *filename* as device.

`bmp("`*`filename`*`")` Opens the Windows bitmap-file *filename* as device.

`windows()` On Windows: opens a graphic window.

`X11()` On Unix/Linux: opens a graphic window.

`quartz()` On Macs: opens a graphic window.

Producing a nice plot rarely works out on the first attempt. Unfortunately there is no way to “undo” a plotting command. The best solution is to type the commands into a script file and then execute the script. Alternatively you might want to store intermediate states before continuing. Here is an example.

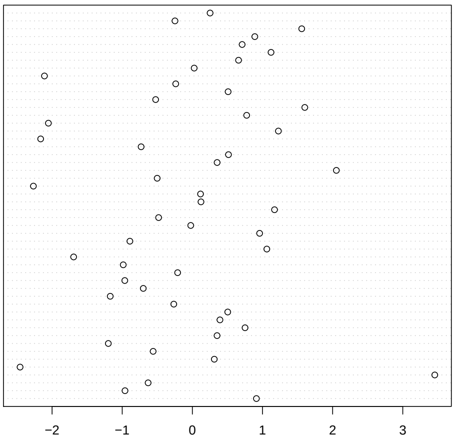
```
  > plot(exp,from=0,to=3)
  > dev.copy(dev.new)     # Save current plot by copying it into a new window
  X11cairo
         3
  > text(2,5,"text at wrong position")
# To "undo" all plotting commands since the last saving, close the current
# plotting window and make the last intermediate state the active plot window.
  > dev.set(dev.prev())
  > dev.copy(dev.new)     # Again save the current plot
  X11cairo
         3
  > text(1,10,"text at good position")
```

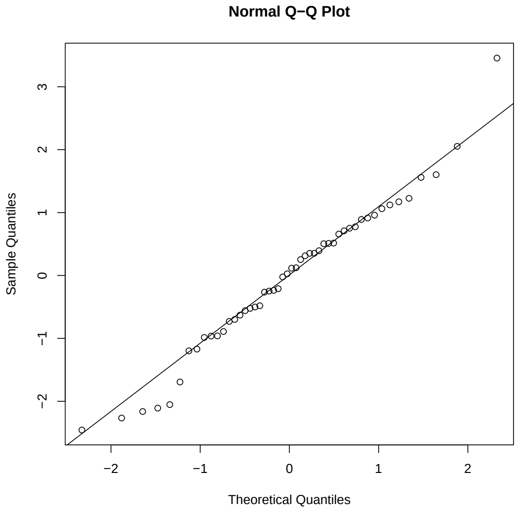
4.6 A list of high-level plotting commands

Further useful high-level plotting functions are listed in the following table.

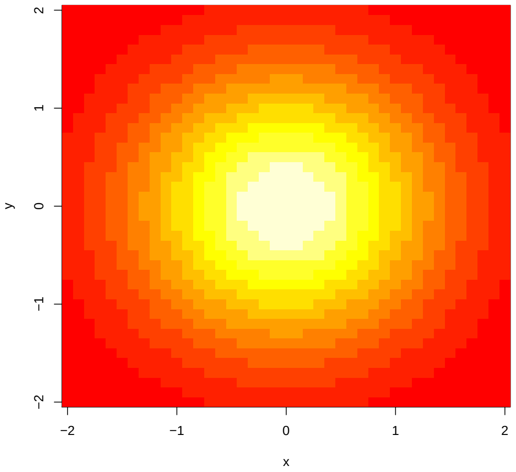
Function	Description
barplot()	Visualizes a vector with bars
boxplot()	Box- and whisker plot
contour()	The contour of a surface is plotted in 2D
coplot()	Conditioning-Plots
dotchart()	Plots the locations of vector elements on the real line
hist()	Histogram
image()	3D-data is visualised with colors
mosaicplot()	Plot in form of a mosaic
pairs()	Produces a matrix of scatterplots
persp()	3D-plot of surfaces
pie()	Circular pie-chart
qqplot()	Quantile-quantile plot

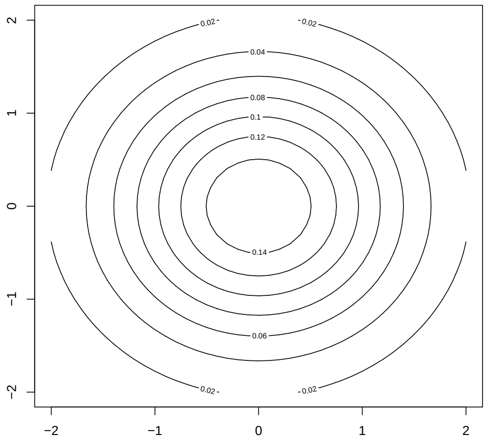
```
> x <- rnorm(50)
> dotchart(x) # plots the points of x
> qqnorm(x)   # plots the sample quantiles of x against the qunatiles of
              # the standard normal distribution
> qqline(x)   # adds a line which passes through the first and third quartiles
```



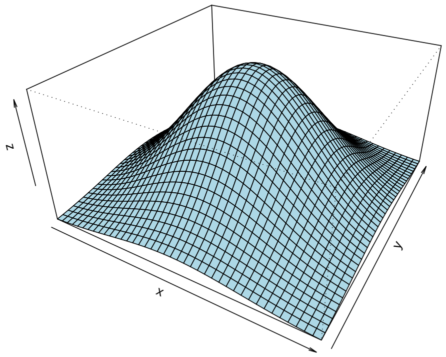


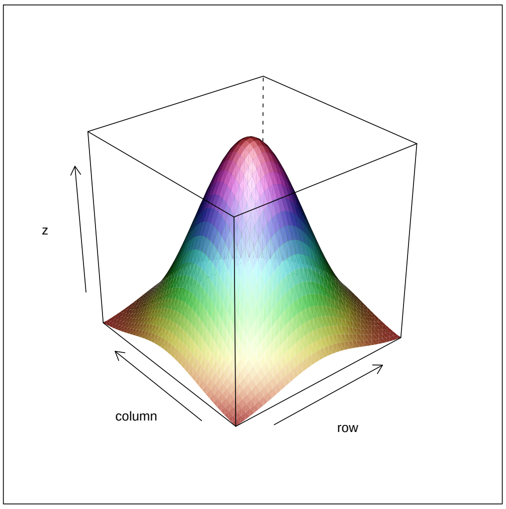
```
> library(mvtnorm)      # library for multivariate normal and t distribution
> x <- seq(from=-2, to=2, by=0.1)
> y <- x
> z <- matrix(nrow=length(x), ncol=length(y), data=0 )
> for ( i in 1:nrow(z) ) for ( j in 1:ncol(z) )
>   z[i,j] <- dmvnorm( cbind( x[i], y[j] ) )
> image(x,y,z)
> contour(x,y,z)
```



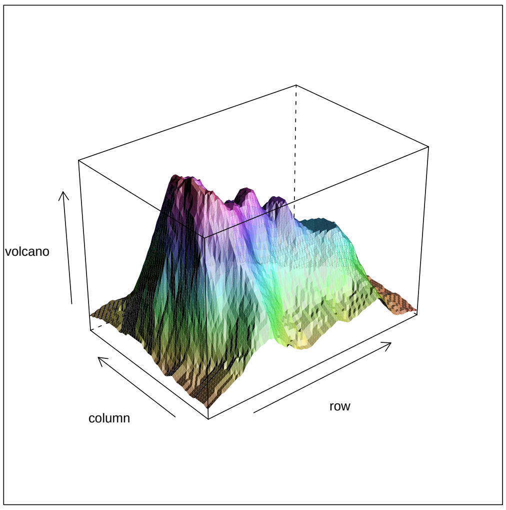


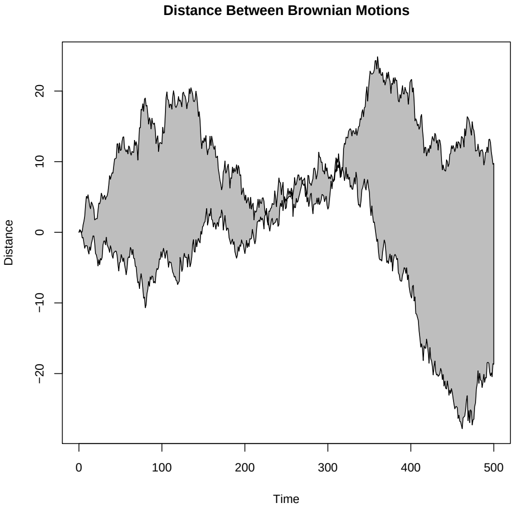
```
> persp(x, y, z, theta = 30, phi = 30, expand = 0.5, col = "lightblue")
> library(lattice) # lattice provides several high-level plotting functions
> trellis.device()
> wireframe(z,shade=TRUE)  # wireframe is in the library lattice
```





```
> data(volcano)  # load volcano data, see ?volcano
> wireframe(volcano,shade=TRUE,apsect=c(61/87,0.4),light.source=c(10,0,10))
## The next example uses cumsum() which sums more and more indeces
> cumsum(1:7)    # 1 1+2 1+2+3 1+2+3+4 1+2+3+4+5 1+2+3+4+5+6 1+2+3+4+5+6+7
[1]  1  3  6 10 15 21 28
> cumsum(c(5,2,9,3))  # 5 5+2 5+2+9 5+2+9+3
[1]  5  7 16 19
## An example showing how to fill between curves.
> par(bg="white")
> n <- 500
> x <- c(0,cumsum(rnorm(n)))
> y <- c(0,cumsum(rnorm(n)))
> xx <- c(0:n, n:0)
> yy <- c(x, rev(y))
> plot(xx, yy, type="n", xlab="Time", ylab="Distance")
> polygon(xx, yy, col="gray")
> title("Distance Between Brownian Motions")
```





Further examples of the graphical capabilities of R can be viewed with

```
> demo(graphics)
> demo(image)
> demo(persp)
> demo(plotmath)
```

There are many more high-level plotting functions in the lattice library (barchart bwplot cloud contourplot densityplot dotplot histogram levelplot parallel piechart qq qqmath rfs splom stripplot tmd wireframe xyplot). If you wish to get an overview of the features of these functions, try the following command.

```
> demo(lattice)
```

`R` has no graphic engine and is therefore not capable of features like rotating 3D figures. To be accurate, there are commands `rotate.cloud()`, `rotate.persp()` and `rotate.wireframe()` in the library 'TeachingDemos'. However these are very slow.

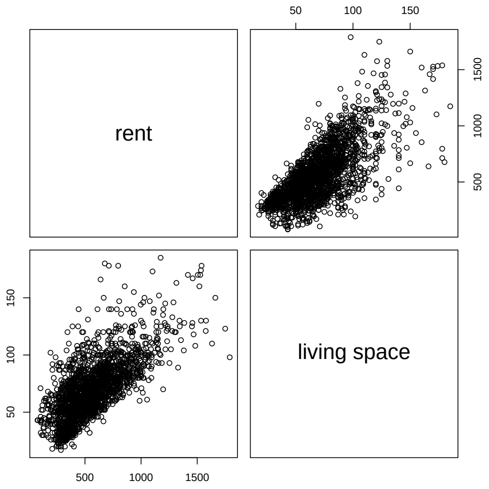
4.7 Displaying multivariate data

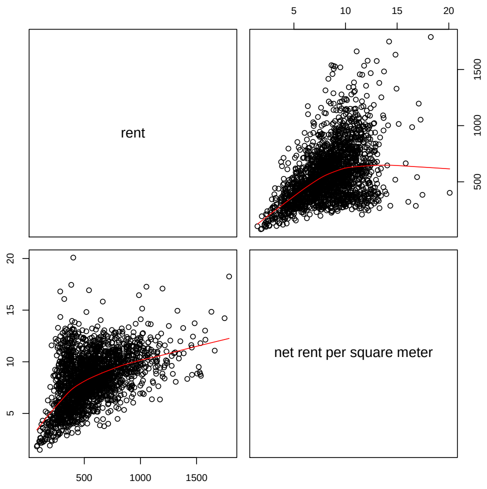
R provides two very useful functions for representing multivariate data. If x is a numeric matrix or data frame, the command

```
> pairs(x)
```

produces a pairwise scatterplot matrix of the variables defined by the columns of x, that is, every column of x is plotted against every other column of x and the resulting plots are arranged in a matrix with plot scales constant over the rows and columns of the matrix.

```
> rent <- read.table("miete03.asc",header=TRUE)
> attach(rent)
> pairs(cbind(nm,wfl), labels=c("rent","living space") )
> pairs(cbind(nm,nmqm), labels=c("rent","net rent per square meter"),
+   panel=panel.smooth )
```





When three or four variables are involved a coplot may be more enlightening. If a and b are numeric vectors and c is a numeric vector or factor object (all of the same length), then the command

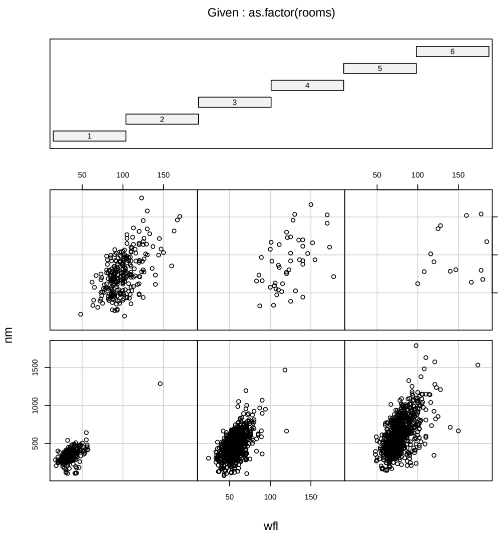
```
> coplot(a ˜ b | c)
```

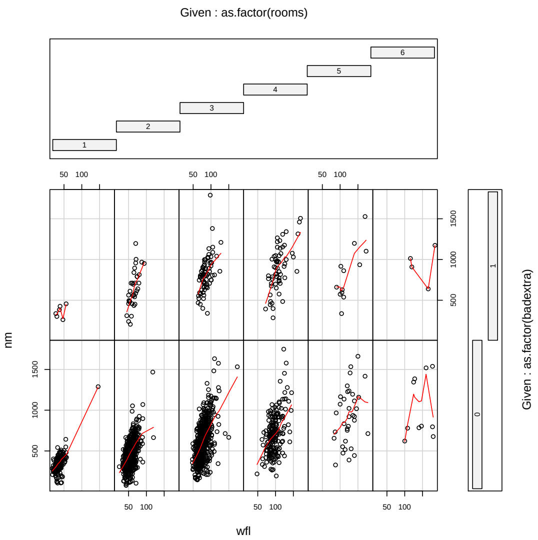
produces a number of scatterplots of a against b for given values of c. If c is a factor, this simply means that a is plotted against b for every level of c. When c is numeric, it is divided into a number of conditioning intervals and for each interval a is plotted against b for values of c within the interval. See ?coplot for more Details. You can also use two given variables with a command like

```
> coplot(a ˜ b | c * d)
```

which produces scatterplots of a against b for every joint conditioning interval of c and d. The coplot() and pairs() function both take an argument `panel=` which can be used to customize the type of plot which appears in each panel. The default is points() to produce a scatterplot but by supplying some other low-level graphics function of two vectors x and y as the value of `panel=` you can produce any type of plot you wish. An example panel function useful for coplots is panel.smooth.

```
> coplot( nm˜wfl | as.factor(rooms) )
> coplot( nm˜wfl | as.factor(rooms)*as.factor(badextra), panel=panel.smooth )
```





4.8 Arguments to high-level plotting functions

High-level plotting functions take a large number of arguments. The list of arguments is included in the help page of par(), see `?par`. For convenience this list is included below. You are not expected to learn these options. Reading through it, however, is recommended.

’add’ Forces the function to act as a low-level graphics function, superimposing the plot on the current plot (does not work reliably).

’adj’ The value of ’adj’ determines the way in which text strings are justified in ’text’, ’mtext’ and ’title’. A value of ’0’ produces left-justified text, ’0.5’ (the default) centred text and ’1’ right-justified text. (Any value in [0, 1] is allowed, and on most devices values outside that interval will also work.) Note that the ’adj’ argument of ’text’ also allows ’adj = c(x, y)’ for different adjustment in x- and y- directions. Note that whereas for ’text’ it refers to positioning of text about a point, for ’mtext’ and ’title’ it controls placement within the plot or device region.

’ann’ If set to ’FALSE’, high-level plotting functions calling ’plot.default’ do not annotate the plots they produce with axis titles and overall titles. The default is to do annotation.

’ask’ logical. If ’TRUE’ (and the R session is interactive) the user is asked for input, before a new figure is drawn. As this applies to the device, it also affects output by packages ’grid’ and ’lattice’. It can be set even on non-screen devices but may have no effect there.

’bg’ The color to be used for the background of the device region. When called from ’par()’ it also sets ’new=FALSE’. See `?colours()` for suitable values. For many devices the initial value is set from the ’bg’ argument of the device, and for the rest it is normally ’"white"’.

Note that some graphics functions such as ’plot.default’ and ’points’ have an _argument_ of this name with a different meaning.

’bty’ A character string which determined the type of ’box’ which is drawn about plots. If ’bty’ is one of ’"o"’ (the default), ’"l"’, ’"7"’, ’"c"’, ’"u"’, or ’"]"’ the resulting box resembles the corresponding upper case letter. A value of ’"n"’ suppresses the box.

’cex’ A numerical value giving the amount by which plotting text and symbols should be magnified relative to the default. Note that some graphics functions such as ’plot.default’ have an _argument_ of this name which _multiplies_ this graphical parameter, and some functions such as ’points’ accept a vector of values which are recycled. Other uses will take just the first value if a vector of length greater than one is supplied.

This starts as ’1’ when a device is opened, and is reset when the layout is changed, e.g. by setting ’mfrow’.

’cex.axis’ The magnification to be used for axis annotation relative to the current setting of ’cex’.

’cex.lab’ The magnification to be used for x and y labels relative to the current setting of ’cex’.

’cex.main’ The magnification to be used for main titles relative to the current setting of ’cex’.

’cex.sub’ The magnification to be used for sub-titles relative to the current setting of ’cex’.

’cin’ Character size ’(width, height)’ in inches. These are the same measurements as ’cra’, expressed in different units.

’col’ A specification for the default plotting color. For a list of available colours, see `?colours()` or `?colors()`. (Some functions such as ’lines’ accept a vector of values which are recycled. Other uses will take just the first value if a vector of length greater than one is supplied.)

’col.axis’ The color to be used for axis annotation. Defaults to ’”black”’.

’col.lab’ The color to be used for x and y labels. Defaults to ’”black”’.

’col.main’ The color to be used for plot main titles. Defaults to ’”black”’.

’col.sub’ The color to be used for plot sub-titles. Defaults to ’”black”’.

’cra’ Size of default character ’(width, height)’ in ’rasters’ (pixels). Some devices have no concept of pixels and so assume an arbitrary pixel size, usually 1/72 inch. These are the same measurements as ’cin’, expressed in different units.

’crt’ A numerical value specifying (in degrees) how single characters should be rotated. It is unwise to expect values other than multiples of 90 to work. Compare with ’srt’ which does string rotation.

’csi’ Height of (default-sized) characters in inches. The same as ’par(”cin”)[2]’.

’cxy’ Size of default character ’(width, height)’ in user coordinate units.
’par(”cxy”)’ is ’par(”cin”)/par(”pin”)’ scaled to user coordinates. Note that ’c(strwidth(ch), strheight(ch))’ for a given string ’ch’ is usually much more precise.

’din’ The device dimensions, ’(width,height)’, in inches.

’err’ (_Unimplemented_; R is silent when points outside the plot region are _not_ plotted.) The degree of error reporting desired.

’family’ The name of a font family for drawing text. The maximum allowed length is 200 bytes. This name gets mapped by each graphics device to a device-specific font description. The default value is ’””’ which means that the default device fonts will be used (and what those are should be listed on the help page for the device). Standard values are ’”serif”’, ’”sans”’ and ’”mono”’, and the Hershey font families are also available. (Different devices may define others, and some devices will ignore this setting completely.) This can be specified inline for ’text’.

’fg’ The color to be used for the foreground of plots. This is the default color used for things like axes and boxes around plots. When called from ’par()’ this also sets parameter ’col’ to the same value. See `?colours()`. A few devices have an argument to set the initial value, which is otherwise ’”black”’.

’fig’ A numerical vector of the form ’c(x1, x2, y1, y2)’ which gives the (NDC) coordinates of the figure region in the display region of the device. If you set this, unlike S, you start a new plot, so to add to an existing plot use ’new=TRUE’ as well.

’fin’ The figure region dimensions, ’(width,height)’, in inches. If you set this, unlike S, you start a new plot.

’font’ An integer which specifies which font to use for text. If possible, device drivers arrange so that 1 corresponds to plain text (the default), 2 to bold face, 3 to italic and 4 to bold italic. Also, font 5 is expected to be the symbol font, in Adobe symbol encoding. On some devices font families can be selected by ’family’ to choose different sets of 5 fonts.

’font.axis’ The font to be used for axis annotation.

’font.lab’ The font to be used for x and y labels.

’font.main’ The font to be used for plot main titles.

’font.sub’ The font to be used for plot sub-titles.

’lab’ A numerical vector of the form ’c(x, y, len)’ which modifies the default way that axes are annotated. The values of ’x’ and ’y’ give the (approximate) number of tickmarks on the x and y axes and ’len’ specifies the label length. The default is ’c(5, 5, 7)’. Note that this only affects the way the parameters ’xaxp’ and ’yaxp’ are set when the user coordinate system is set up, and is not consulted when axes are drawn. ’len’ _is unimplemented_ in R.

’las’ numeric in 0,1,2,3; the style of axis labels.

0: always parallel to the axis [_default_],

1: always horizontal,

2: always perpendicular to the axis,

3: always vertical.

Also supported by ’mtext’. Note that other string/character rotation (via argument ’srt’ to ’par’) does _not_ affect the axis labels.

’lend’ The line end style. This can be specified as an integer or string:

’0’ and ’”round”’ mean rounded line caps [_default_];

’1’ and ’”butt”’ mean butt line caps;

’2’ and ’”square”’ mean square line caps.

’lheight’ The line height multiplier. The height of a line of text (used to vertically space multi-line text) is found by multiplying the character height both by the current character expansion and by the line height multiplier. Default value is 1. Used in ’text’ and ’strheight’.

’ljoin’ The line join style. This can be specified as an integer or string:

’0’ and ’”round”’ mean rounded line joins [_default_];

’1’ and ’”mitre”’ mean mitred line joins;

’2’ and ’”bevel”’ mean bevelled line joins.

’lmitre’ The line mitre limit. This controls when mitred line joins are automatically converted into bevelled line joins. The value must be larger than 1 and the default is 10. Not all devices will honour this setting.

’lty’ The line type. Line types can either be specified as an integer (0=blank, 1=solid (default), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash) or as one of the character strings ’”blank”’, ’”solid”’, ’”dashed”’, ’”dotted”’, ’”dotdash”’, ’”longdash”’, or ’”twodash”’, where ’”blank”’ uses ’invisible lines’ (i.e., does not draw them).

Alternatively, a string of up to 8 characters (from ’c(1:9, ”A”:”F”)’) may be given, giving the length of line segments which are alternatively drawn and skipped.

Some functions such as ’lines’ accept a vector of values which are recycled. Other uses will take just the first value if a vector of length greater than one is supplied.

’lwd’ The line width, a _positive_ number, defaulting to ’1’. The interpretation is device-specific, and some devices do not implement line widths less than one. (See the help on the device for details of the interpretation.)

Some functions such as ’lines’ accept a vector of values which are recycled. Other uses will take just the first value if a vector of length greater than one is supplied.

’mai’ A numerical vector of the form ’c(bottom, left, top, right)’ which gives the margin size specified in inches.

’main’ Figure title, placed at the top of the plot in a large font.

’mar’ A numerical vector of the form ’c(bottom, left, top, right)’ which gives the number of lines of margin to be specified on the four sides of the plot. The default is ’c(5, 4, 4, 2) + 0.1’.

’mex’ ’mex’ is a character size expansion factor which is used to describe coordinates in the margins of plots. Note that this does not change the font size, rather specifies the size of font (as a multiple of ’csi’) used to convert between ’mar’ and ’mai’, and between ’oma’ and ’omi’.

This starts as ’1’ when the device is opened, and is reset when the layout is changed (alongside resetting ’cex’).

’mfcol’

’mfrow’ A vector of the form ’c(nr, nc)’. Subsequent figures will be drawn in an ’nr’-by-’nc’ array on the device by _columns_ (’mfcol’), or _rows_ (’mfrow’), respectively.

In a layout with exactly two rows and columns the base value of ’”cex”’ is reduced by a factor of 0.83: if there are three or more of either rows or columns, the reduction factor is 0.66. Setting a layout resets the base value of ’cex’ and that of ’mex’ to ’1’.

If either of these is queried it will give the current layout, so querying cannot tell you the order the array will be filled.

Consider the alternatives, ’layout’ and ’split.screen’.

’mfg’ A numerical vector of the form ’c(i, j)’ where ’i’ and ’j’ indicate which figure in an array of figures is to be drawn next (if setting) or is being drawn (if enquiring). The array must already have been set by ’mfcol’ or ’mfrow’.

For compatibility with S, the form ’c(i, j, nr, nc)’ is also accepted, when ’nr’ and ’nc’ should be the current number of rows and number of columns. Mismatches will be ignored, with a warning.

'mgp' The margin line (in 'mex' units) for the axis title, axis labels and axis line. Note that 'mgp[1]' affects 'title' whereas 'mgp[2:3]' affect 'axis'. The default is 'c(3, 1, 0)'.

'new' logical, defaulting to 'FALSE'. If set to 'TRUE', the next high-level plotting command (actually 'plot.new') should _not clean_ the frame before drawing _as if it was on a *_new_* device_. It is an error (ignored with a warning) to try to use 'new=TRUE' on a device that does not currently contain a high-level plot.

'oma' A vector of the form 'c(bottom, left, top, right)' giving the size of the outer margins in lines of text.

'omd' A vector of the form 'c(x1, x2, y1, y2)' giving the region _inside_ outer margins in NDC (= normalised device coordinates), i.e., as fraction (in [0,1]) of the device region.

'omi' A vector of the form 'c(bottom, left, top, right)' giving the size of the outer margins in inches.

'pch' Either an integer specifying a symbol or a single character to be used as the default in plotting points. See `?points` for possible values and their interpretation. Note that only integers and single-character strings can be set as a graphics parameter (and not 'NA' nor 'NULL').

'pin' The current plot dimensions, '(width,height)', in inches.

'plt' A vector of the form 'c(x1, x2, y1, y2)' giving the coordinates of the plot region as fractions of the current figure region.

'ps' integer; the point size of text (but not symbols). Unlike the 'pointsize' argument of most devices, this does not change the relationship between 'mar' and 'mai' (nor 'oma' and 'omi').

What is meant by 'point size' is device-specific, but most devices mean a multiple of 1bp, that is 1/72 of an inch.

'pty' A character specifying the type of plot region to be used; '"s"' generates a square plotting region and '"m"' generates the maximal plotting region.

'smo' (_Unimplemented_) a value which indicates how smooth circles and circular arcs should be.

'srt' The string rotation in degrees. See the comment about 'crt'. Only supported by 'text'.

'sub' Sub-title, placed just below the x-axis in a smaller font.

'tck' The length of tick marks as a fraction of the smaller of the width or height of the plotting region. If `tck >= 0.5` it is interpreted as a fraction of the relevant side, so if `tck =1` grid lines are drawn. The default setting (`tck = NA`) is to use `tcl = -0.5`.

'tcl' The length of tick marks as a fraction of the height of a line of text. The default value is '-0.5'; setting 'tcl = NA' sets 'tck = -0.01' which is S' default.

'type' The `type=` argument controls the type of plot produced, as follows:

`type="p"` Plot individual points (the default)

`type="l"` Plot lines

`type="b"` Plot points connected by lines (both)

`type="o"` Plot points overlaid by lines

`type="h"` Plot vertical lines from points to the zero axis (high-density) `type="s"`

`type="S"` Step-function plots. In the first form, the top of the vertical defines the point; in the second, the bottom.

`type="n"` No plotting at all. However axes are still drawn (by default) and the coordinate system is set up according to the data. Ideal for creating plots with subsequent low-level graphics functions.

'usr' A vector of the form 'c(x1, x2, y1, y2)' giving the extremes of the user coordinates of the plotting region. When a logarithmic scale is in use (i.e., `par("xlog")` is true, see below), then the x-limits will be `10^par("usr")[1:2]`. Similarly for the y-axis.

'xaxp' A vector of the form 'c(x1, x2, n)' giving the coordinates of the extreme tick marks and the number of intervals between tick-marks when `par("xlog")` is false. Otherwise, when _log_ coordinates are active, the three values have a different meaning: For a small range, 'n' is _negative_, and the ticks are as in the linear case, otherwise, 'n' is in '1:3', specifying a case number, and 'x1' and 'x2' are the lowest and highest power of 10 inside the user coordinates, `10^par("usr")[1:2]`. (The '"usr"' coordinates are log10-transformed here!)

n=1 will produce tick marks at `10^j` for integer j,

n=2 gives marks k `10^j` with k in 1, 5,

n=3 gives marks k `10^j` with k in 1, 2, 5.

See 'axTicks()' for a pure R implementation of this.

This parameter is reset when a user coordinate system is set up, for example by starting a new page or by calling 'plot.window' or setting 'par("usr")': 'n' is taken from 'par("lab")'. It affects the default behaviour of subsequent calls to 'axis' for sides 1 or 3.

'xaxs' The style of axis interval calculation to be used for the x-axis. Possible values are '"r"', '"i"', '"e"', '"s"', '"d"'. The styles are generally controlled by the range of data or 'xlim', if given. Style '"r"' (regular) first extends the data range by 4 percent at each end and then finds an axis with pretty labels that fits within the extended range. Style '"i"' (internal) just finds an axis with pretty labels that fits within the original data range. Style '"s"' (standard) finds an axis with pretty labels within which the original data range fits. Style '"e"' (extended) is like style '"s"', except that it is also ensures that there is room for plotting symbols within the bounding box. Style '"d"' (direct) specifies that the current axis should be used on subsequent plots. (_Only '"r"' and '"i"' styles are currently implemented_)

'xaxt' A character which specifies the x axis type. Specifying '"n"' suppresses plotting of the axis. The standard value is '"s"': for compatibility with S values '"l"' and '"t"' are accepted but are equivalent to '"s"': any value other than '"n"' implies plotting.

'xlab' Axis labels for the x axes.

'xlog' A logical value (see 'log' in 'plot.default'). If 'TRUE', a logarithmic scale is in use (e.g., after 'plot(*, log = "x")'). For a new device, it defaults to 'FALSE', i.e., linear scale.

'xpd' A logical value or 'NA'. If 'FALSE', all plotting is clipped to the plot region, if 'TRUE', all plotting is clipped to the figure region, and if 'NA', all plotting is clipped to the device region. See also 'clip'.

'yaxp' A vector of the form 'c(y1, y2, n)' giving the coordinates of the extreme tick marks and the number of intervals between tick-marks unless for log coordinates, see 'xaxp' above.

'yaxs' The style of axis interval calculation to be used for the y-axis. See 'xaxs' above.

'yaxt' A character which specifies the y axis type. Specifying '"n"' suppresses plotting.

'ylab' Axis labels for the x axes.

'ylog' A logical value; see 'xlog' above.

5 Some statistical tests

5.1 Theory of statistical tests

- You want to support a hypothesis (e.g. a medication is effective)
- Assuming that measurements are subject to randomness, some pessimist could always claim that the signal in your data is purely random ("that all 20 patients recovered might have happened by chance")
- In order to refute the pessimist, assume that he is right and show that the observed data are very unlikely under this assumption.

In statistical language, the opinion of the pessimist is called **null hypothesis** which generally claims that "observation is due to randomness" and is formulated more precisely for every test, e.g., H_0: the medication has the same effect as placebo. Procedure of a statistical test:

- Formulate the null hypothesis, e.g., H_0: the medication has the same effect as placebo.
- Show that the observation and everything more 'extreme' is sufficiently unlikely under this null hypothesis. Scientists have agreed that it suffices that this probability is at most 5%.
- This refutes the pessimist. Statistical language: We reject the null hypothesis on the significance level 5%.

The probability of the observation and everything more 'extreme' under the null hypothesis is called **p-value**. More formally

$$p - \text{value} = \mathbb{P}\left(\text{ observation and everything more 'extreme' } | H_0 \text{ is true } \right).$$

So a p-value of 2% means that if the pessimist is right, then only 2 out of 100 experiments result in such an observation (on average). Note that we did not disprove the pessimist. The null hypotheis could still be true. However, if the null hypothesis *is* true, then it is unlikely to get such an observation.

If the p-value is above the significance level, p-value= 8% say, then we cannot refute the null hypothesis on the 5%-level. Of course this does in no way 'prove' the null hypothesis. Not being able to reject the null hypothesis is rather like being undecided, having no opinion.

The following two subsections give two important tests.

5.2 Test for a difference in mean: t-test

What is given? Independent observations $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_m)$.

Null hypothesis: x and y are samples from distributions having the same mean.

Test: t-test

R command: `t.test(` *x*`,` *y*`)`

Idea of the test: If the sample means are too far apart, then reject the null hypothesis.

The t-test is an approximative test, the test statistic is only approximatively t-distributed. Fortunately, the test rather robust. The test works also for small sample sizes quite fine. The t-test is only sensible to the violation of independence. So if the samples are dependent, then be careful.

Example: In contrast to the common opinion, there are not only green marsians but also red and blue marsians. The file 'mars.txt' contains the height (in cm) and the color of all 42 marsians which have been found in the last 50 years. We wish to support the hypothesis that the height of green marsians is different on average from the height of blue marsians.

```
> mars <- read.table("mars.txt",header=TRUE)
> head(mars)
      size color
1 65.67974   red
2 65.90436   red
3 67.34730   red
4 60.42924   red
5 55.34526   red
6 62.85024   red
> attach(mars)
> t.test(size[color=="green"],size[color=="blue"])

        Two Sample t-test

data:  size[color == "green"] and size[color == "blue"]
t = -3.4244, df = 19.419, p-value = 0.002775
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -16.875514  -4.083647
sample estimates:
mean of x mean of y
 60.86840  71.34798
```

Answer: We reject the null hypothesis that green and blue marsians have the same height on average (5% significance level). Here we used an **unpaired t-test**, as there is no dependence between the two samples.

Here is an example for a **paired t-test**. We are given the wear of shoes of materials A and B for one foot of each of ten boys. The two samples are now correlated through the boy who wore the respective shoe. Some boys cause higher wear and some boys smaller wear. Thus we need to apply the paired t-test.

```
> data(shoes,package='MASS')
> attach(shoes)
> head(shoes)
$A
[1] 13.2  8.2 10.9 14.3 10.7  6.6  9.5 10.8  8.8 13.3

$B
[1] 14.0  8.8 11.2 14.2 11.8  6.4  9.8 11.3  9.3 13.6
```

```
> t.test(A,B,paired=TRUE)

        Paired t-test

data:  A and B
t = -3.3489, df = 9, p-value = 0.008539
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.6869539 -0.1330461
 sample estimates:
 mean of the differences
                  -0.41
```

Answer: We reject the null hypothesis that the two materials A and B are equally good on average (5% significance level).

5.3 Test for dependence

It depends on the type of variable which test to use:

- variables have nominal values (no ordering, e.g., eye color or gender)
- variables have ordinal values (values are ordered but not continuous, e.g. result of dice, age in years $\in \mathbb{N}$)
- variables have continous values (any value in some interval is possible, e.g. body height, age $\in [0, \infty)$)

5.3.1 Nominal variables (count data)

Paired observations $(X_1, Y_1), \ldots, (X_n, Y_n)$. Arrange observations as contingency table. Example: X = eye color, Y = hair color

	blond	brown	black	total
blue	47	42	8	97
brown	3	60	33	96
green	8	15	3	26
total	58	117	44	219

What is given? Pairwise observations $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$

Null hypothesis: x and y are independent

Test: χ^2-test for independence

R command: `chisq.test(` *`x`*`,` *`y`*`)` or `chisq.test(` *`contingency.table`*`)`

Idea of the test: Calculate the expected abundancies under the assumption of independence. If the observed abundancies deviate too much from the expected abundancies, then reject the null hypothesis.

The χ^2 test for independence is an approximative test, the test statistic is only approximatively χ^2-distributed. This test should only be applied, if the following condition is statisfied. Let n_{kl}

be the entries of the contingency table. Let $n_{k\cdot} := \sum_l n_{kl}$ be the row sums, let $n_{\cdot l} := \sum_k n_{kl}$ be the column sums and let $n := \sum_k \sum_l n_{kl}$ be the total sum. Then the expected abundancies are $n^*_{kl} := \frac{n_{k\cdot} \cdot n_{\cdot l}}{n}$.

Rule of thumb for χ^2-test: All expected abundancies should be bigger than 1 and 80% of all expected abundancies should be bigger than 5.

Example:

```
> contingency <- matrix( c(47,3,8,42,60,15,8,33,3), nrow=3 )
> chisq.test(contingency)$expected
         [,1]     [,2]      [,3]
[1,] 25.689498 51.82192 19.488584
[2,] 25.424658 51.28767 19.287671
[3,]  6.885845 13.89041  5.223744
# expected abundancies are all above 5, so we may apply the test
> chisq.test(contingency)

        Pearson's Chi-squared test

data:  contingency
X-squared = 58.5349, df = 4, p-value = 5.892e-12
```

Answer: We reject the null hypothesis that eye color and hair color are independent (5% significance level).

In the special case of 2 × 2 contingency tables, the χ^2-approximation is not needed. Here you should use **Fisher's exact test**.

What is given? Pairwise observations $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$; two categories both for x and y

Null hypothesis: x and y are independent

Test: Fisher's exact test for independence

R command: `fisher.test(` *`x, y`* `)` or `fisher.test(` *`contingency.table`* `)`

Example: Rosen and Jerdee (Influence of sex role stereotype on personnel decisions, *J. Appl. Psych* **59**, 9–14, 1974) let 48 participants of a management course look at personnel files and let them decide whether to advance the person or not. The personnel files where identical except for the gender (24 female, 24 male). The result was

	female	male
advancement	14	21
no advancement	10	3

```
> table <- matrix( c(14,10,21,3), nrow=2 )
> fisher.test(table)

          Fisher's Exact Test for Count Data
```

```
data:  table
p-value = 0.04899
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.03105031 0.99446037
 sample estimates:
 odds ratio
  0.2069884
```

Answer: We reject the null hypothesis that gender of the personnel and the decision for advancement are independent (5% significance level).

5.3.2 Continuous variables

Here we assume that the variables could – in principle – take all values of some interval.

What is given? Pairwise observations $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$; all values in some interval are possible

Null hypothesis: x and y are independent

Test: Pearson's correlation test for independence

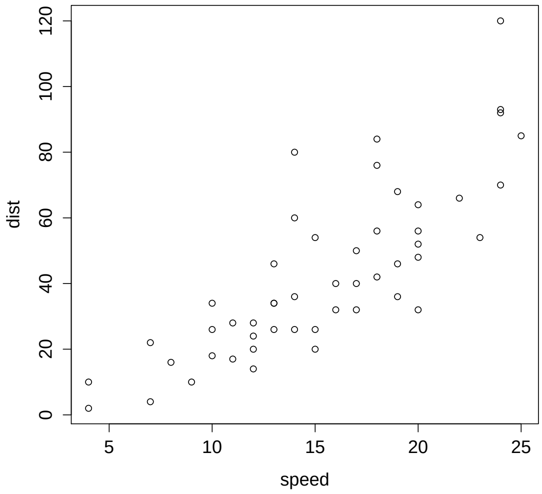
Assumption: x and y are samples from a normal distribution

R command: `cor.test(` *x*`,` *y*`)`

Example: Distance needed to stop (in ft) from a certain speed (mph):

```
> data(cars)    # cars is a dataset in the library 'datasets', see ?cars
> attach(cars)
> str(cars)
> ?cars
> plot(speed,dist)
> cor.test(speed, dist)

Pearson's product-moment correlation

data:  speed and dist
t = 9.464, df = 48, p-value = 1.49e-12
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.6816422 0.8862036
 sample estimates:
      cor
0.8068949
```

Answer: We reject the null hypothesis that 'speed' and 'dist' are independent (5% significance level).



5.3.3 Ordinal variables

Here we assume that the observations can be ordered. In particular we assume that there are no (at least not many) repeated values.

What is given? Pairwise observations $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$; values can be ordered

Null hypothesis: x and y are uncorrelated

Test: Spearman's rank correlation rho

R command: `cor.test(` *x*, *y*, `method="spearman")`

Example: Distance needed to stop (in ft) from a certain speed (mph):

```
> attach(cars)
> cor.test(speed, dist, method="spearman")

Spearman's rank correlation rho

data:  speed and dist
S = 3532.819, p-value = 8.825e-14
alternative hypothesis: true rho is not equal to 0
sample estimates:
      rho
0.8303568

Warnmeldung:
In cor.test.default(speed, dist, method = "spearman") :
  Kann exakte p-Werte bei Bindungen nicht berechnen
```

Answer: We reject the null hypothesis that 'speed' and 'dist' are independent (5% significance level).

5.4 The power of a test

The alternative hypothesis to the null hypothesis is called **alternative** (in symbols: H_1). What you want to show should be part of the alternative. Example:

$$H_0\colon \mu = 0$$
$$H_1\colon \mu \neq 0$$

There are two possible errors:

- A "type I error", also known as "error of the first kind", an "α error" or a "false positive": the error of rejecting the null hypothesis when it is actually true. Example: A test indicates pregnancy although the woman is not pregnant.
- A "type II error", also known as "error of the second kind", an "β error" or a "false negative": the error of failing to reject the null hypothesis when it is actually not true. Example: A test indicates no pregnancy although the woman is in fact pregnant.

	null hypothesis H_0 is true	**alternative H_1 is true**
null hypothesis not rejected	correct	type II error, false negative
null hypothesis rejected	type I error, false positive	correct

If β is the probability of a type II error, then the power of the test is defined as $1-\beta$. If the power of the test is 0, then the null hypothesis will never be rejected even if the alternative is true. As we want to reject the null hypothesis, we would rather prefer a test power close to 1. For this, the alternative $H_1\colon \mu \neq 0$ is bad. The problem is that if the true value μ is extremely close to 0, then the test has no chance to tell the difference from zero. Instead one would rather prefer an alternative like $H_1\colon |\mu| \geq 0.5$.

In general, the test power increases with the sample size. There are functions e.g. `power.t.test()` or `power.fisher.test()` to calculate the minimal sample size needed to achieve a given power of the test. For details, see a statistics course.

5.5 A list of statistical tests in R

Test	R command	Description
t-test	t.test()	Are the true means of two samples different?
F-test	aov()	Are the true means of several samples different? (balanced designs)
Chi-squared test	chisq.test()	test for independence of paired samples (count data)
Fisher's exact test	fisher.test()	test for independence of paired samples (count data)
Spearman's rho, Kendall's tau	cor.test()	test for independence of paired samples
Wilcoxon, Mann-Whitney	wilcox.test()	Is the location parameter the same in two groups?
Kruskal-Wallis rank sum test	kruskal.test()	Is the location parameter the same in each group?
Shapiro-Wilk normality test	shapiro.test()	Is the sample drawn from a normal distribution?
Kolmogorov-Smirnov Test	ks.test()	Are two samples drawn from the same distribution?
Quade-test	quade.test()	Is the location parameter the same in each group? (block designs)
Friedman-test	friedman.test()	Is the location parameter the same in each group? (block designs)
Exact Binomial test	binom.test()	Exact test for the probability of success in a Bernoulli experiment
Variance test	var.test()	test for homogeneity of variances across two groups
Bartlett test	bartlett.test()	test for homogeneity of variances across groups
Fligner-Killeen test	fligner.test()	test for homogeneity of variances across groups
Levene's test	levene.test()	test for homogeneity of variances across groups
Test of equal proportions	prop.test()	Are the probabilities of success the same in several groups?
Kendall's tau, Spearman's rho	cor.test()	test for independence between paired samples
Box-Pierce, Ljung-Box	Box.test()	tests for independence in a time series
Cochran-Mantel-Hanszel test	mantelhaen.test()	Are the (nominal) variables conditionally independent in each stratum?
Mood test of scale	mood.test()	two-sample test for a difference in scale parameters
Ansari-Bradley test	ansari.test()	test for a difference in scale parameters
McNemar's Chi-squared test	mcnemar.test()	test for symmetry of rows and columns in a 2-dimensional contingency table
Mauchly's test of sphericity	mauchly.test()	Is a Wishart-distributed covariance matrix proportional to a given matrix?
Exact Poisson test	poisson.test()	tests for the rate parameter in a Poisson distribution

5.6 Degrees of freedom

The phrase *degrees of freedom* turns up from time to time. As this concept is often unknown, we quickly explain it. Suppose you are free to choose five values x_1, x_2, x_3, x_4, x_5. Then your degree of freedom is 5. Similarly there are five degrees of freedom in the vector

$$x <\!- c(x_1, x_2, x_3, x_4, x_5).$$

Now let us consider the vector

$$v <- \; x - \text{mean}(x) = c(x_1, x_2, x_3, x_4, x_5) - \text{mean}(x).$$

Here you may freely choose v_1, v_2, v_3, v_4. The value of v_5, however, is then fixed. We know already that the mean of v is equal to mean$(v) = 0$. So the fifth element of v is determined through $v_1 + v_2 + v_3 + v_4 + v_5 = 0$.

More generally if x is a vector of length n, then there are $n - 1$ degrees of freedom in the vector $x - \text{mean}(x)$. A different argument for this is that there is one parameter estimated from x namely mean(x). This reduces the degrees of freedom by one. Hence there remain $n - 1$ degrees of freedom. More generally still, we propose a formal definition of degrees of freedom:

degrees of freedom of a sample =
the sample size minus the number of parameters
estimated from the sample.

6 Programming in R

6.1 Conditional execution: if() and ifelse()

Syntax:

if (*condition*) { *commands1* }
if (*condition*) { *commands1* } else { *commands2* }
ifelse (*conditions_vector*, *yes_vector*, *no_vector*)

The commmand if() evaluates '*commands1*' if the logical expression '*condition*' returns TRUE. Here '*commands1*' is a single command or a sequence of commands separated with ';'. The comand if()else evaluates '*commands1*' if the logical expression '*condition*' returns TRUE, otherwise it evaluates '*commands2*'. The command ifelse() returns a vector of the same length as '*conditions_vector*' with elements selected from either '*yes_vector*' or '*no_vector*' depending on whether the element of '*conditions_vector*' is TRUE or FALSE. Here are examples:

```
> x <- 4
> if ( x == 5 ) { x <- x+1 } else { x <- x*2 }
> x
[1] 8
> if ( x != 5 & x>3 ) { x <- x+1 ; 17+2 } else { x <- x*2 ; 21+5 }
[1] 19
> x
[1] 9
> y <- 1:10
> ifelse( y<6, y^2, y-1 )
 [1]  1  4  9 16 25  5  6  7  8  9
> z <- 6:-3
> sqrt(z)                              # Produces a warning
 [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000      NaN
 [9]      NaN      NaN
Warning message:
In sqrt(z) : NaNs
> sqrt( ifelse( z>=0, z, NA ) )      # No warning
 [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000       NA
 [9]       NA       NA
```

6.2 Loops: for(), while() and repeat()

Syntax:

for (*var* in *set*) { *commands* }
while (*condition*) { *commands* }
repeat { *commands* }

The object *set* is a vector, *commands* is a single command or a sequence of commands and *var* is a variable which may be used in *commands*. The command `for()` is the R version of 'for each element in the set do ...'. The command `while()` is the R version of 'as long as the condition is TRUE do ...'. The command `repeat()` is the R version of 'repeat until I say break'. The command '`break`' stops any loop; control is then transferred to the first statement outside the loop. The command '`next`' halts the processing of the current iteration and advances the looping index. Here are examples:

```
> x <- 0
> for ( i in 1:5 ) { if (i==3) { next } ; x <- x + i }
> x               # i=3 is skipped, so x <- 1+2+4+5
[1] 12
> y <- 1; j <- 1
> while ( y < 12 & j < 8 ) { y <- y*2 ; j <- j + 1}
> y; j
[1] 16
[1] 4
> z <- 3
> repeat { z<- z^2; if ( z>100 ) { break }; print(z)}
[1] 9
[1] 81
> z              # the loop stopped after 81^2, so z==81^2
[1] 6561
```

6.3 Examples

Let us approximate the mean of a dice. We use the command `sample()` for simulating a dice. The command `sample(v,n,replace=TRUE)` produces a sample (random draw) of length n from the vector v with replacement.

```
> x<-sample(1:6,1000,replace=TRUE)
> mean(x)
[1] 3.469
```

The result is close to the true value 3.5.

Let us now check that the t-test respects the significance level, that is, let us check that the p-value of the t-test is below 0.05 in roughly 5% of the cases.

```
> counter <- 0
> for( i in 1:10000) {
+   if( t.test(rnorm(100),rnorm(100))$p.value <= 0.05 ) {
+     counter <- counter + 1
+   }
+ }
> counter/10000
[1] 0.0497
>
```

Indeed, the probability of the error of the first kind is close to 5%. Let's repeat that with a shorter sample size:

```
> counter <- 0
> for( i in 1:10000) {
+   if( t.test(rnorm(10),rnorm(10))$p.value <= 0.01 ) {
+     counter <- counter + 1
+   }
+ }
> counter/10000
[1] 0.009
```

Again reasonably close to the theoretical value 1%.

6.4 Executing commands from a script

Instead of typing commands in R it is often more convenient to type the commands into a file and then execute the file with R. Files with R commands are called R scripts and are usually given the extension '.R' (or '.r') although this is not necessary. You execute an R script with the command source(). The argument for source() is the filename of the script (the name must be quoted). Let 'C:/Documents/R/myscript.R' be the following R script:

```
cat("This is the script 'myscript.R'\n")
5+3
print(4+1)
x <- 7
if ( x == 3 )
{
  cat("The value of x is equal to 3\n")
} else {
  cat("The value of x is not equal to 3 but equal to",x,"\n")
}
for ( i in list("Anton","Berta","Casper") )
{
  cat(i)
  cat(" ")
}
cat("\n")
```

Now we execute the script.

```
> source("C:/Documents/R/myscript.R")
This is the script 'myscript.R'
[1] 5
The value of x is not equal to 3 but equal to 7
Anton Berta Casper
```

If you wish to avoid entering the pathname in the command source() and if 'myscript.R' is in the current working directory, then it suffices to say source("myscript.R"). If you are unsure, what the current working directory is, use the command getwd(). You may change the current working directory with setwd().

```
> source("myscript.R")
Error in file(file, "r", encoding = encoding) :
cannot open connection
Additional warning:
In file(file, "r", encoding = encoding) :
cannot open file 'myscript.R': No such file or directory
# myscript.R does not exist? Probably wrong working directory. Let's check.
> getwd()
[1] "C:/Documents/")
> setwd("C:/Documents/R")    # We need to change the working directory
> getwd()
[1] "C:/Documents/R")
> dir()                      # Show all files in the current directory
[1] "somepdffile.pdf" "myscript.R" "otherscript.R"
```

```
> source("myscript.R")         # Now it works
This is the script 'myscript.R'
[1]
The value of x is not equal to 3 but equal to 7
Anton Berta Casper
```

Another way to execute the script is in Unix and Macintosh to enter 'Rscript myscript.R' on the shell command line.

6.5 Writing your own functions

Syntax:

myfun <− function (*arg1*, *arg2*, . . .) { *commands* }

This defines a function with name *myfun*. The function arguments *arg1*, *arg2*, . . . are then processed with the sequence of commands *commands*. Use the command return() for returning a value. If you need to return several values put them into a list and return the list.

Being defined the function can be called as *myfun(expr1,expr2,...)* where *expr1*, *expr2*, ... are expressions whose values are assigned to the respective argument in the definition of the function.

```
> se <- function(x)
+ {
+   y<-sqrt(var(x)/length(x))
+   return(y)
+ }
> se(1:4)
[1] 0.6454972
>
> se("wrong type of argument")
[1] NA
Warning message:
In var(x) : NAs introduced by coercion
```

In case of an error (e.g. wrong type of arguments), use the command stop(*"errormsg"*) to stop the execution of the function and to return the error message '*errormsg*'. Improved version:

```
> se <- function(x)
+ {
+   if (is.numeric(x)!=TRUE)
+   {
+     stop("need numeric data")
+   }
+   y<-sqrt(var(x)/length(x))
+ }
> se(1:4)
[1] 0.6454972
> se("wrong type of argument")
Error in se("wrong type of argument") : need numeric data
> se(c(NA,1:4))
[1] NA
> sum(c(NA,1:4),na.rm=TRUE)
[1] 10
```

In many cases arguments can be given commonly appropriate default values, in which case they may be omitted from the call when the defaults are appropriate. Improved version:

```
> se <- function(x,na.rm=FALSE)
+ {
+   if (is.numeric(x)!=TRUE)
+   {
+     stop("need numeric data")
+   }
+   if (na.rm==TRUE)
+   {
+     x<-x[is.na(x)==FALSE]
+   }
+   y<-sqrt(var(x)/length(x))
+ }
> se(c(NA,1:4))
[1] NA
> se(c(NA,1:4),na.rm=TRUE)
[1] 0.6454972
> se(c(NA,1:4),TRUE)
[1] 0.6454972
> se(na.rm=TRUE,c(NA,1:4))
[1] 0.6454972
> se(TRUE,c(NA,1:4))
Error in se(TRUE,c(NA,1:4)) : argument is required to be numeric
```

The arguments in the call `se(c(NA,1:4),TRUE)` are assigned to the variables of `se()` in the order specified by the definition of `se()`. Remembering this order of arguments is often inconvenient. The order of arguments does not matter if you reference by name, that is, if you specify arguments in the form 'name=*object*'. Furthermore there may be both unnamed and named arguments.

Another feature of R is the '. . . ' argument. Thereby one can pass on arguments from one function to another function.

```
> se.sq <- function(x, ... )
+ {
+   y <- se( x, ... )
+   return(y^2)
+ }
```

Whatever is assigned to the '...' argument is put into the argument list of `se()`.

```
> se.sq(1:4)
[1] 0.4166667
> se.sq(c(NA,1:4))
[1] NA
> se.sq(c(NA,1:4),na.rm=TRUE)
[1] 0.4166667
```

If you want to return several values, then return them as vector or as list. Here is an example which returns the confidence interval of the mean of a sample based on the normal approximation.

```
> ci.norm <- function(x,conf=0.95)
```

```
+ {
+   q <- qnorm(1-(1-conf)/2)
+   return( list(lower=mean(x)-q*se(x),upper=mean(x)+q*se(x)) )
+ }
> ci.norm(rnorm(100))
$lower
[1] -0.1499551

$upper
[1] 0.2754680

> ci.norm(rnorm(100),conf=0.99)
$lower
[1] -0.1673693

$upper
[1] 0.2443276
```

Note that many commands in R (e.g. `mean()`, `var()`, `median()`) are functions written with `function()`. Here is a modified version of the command `median()`:

```
mymedian <- function(x, na.rm=FALSE)
{
  if( mode(x) != "numeric" )
  {
    stop("need numeric data")
  }
  if( na.rm )
  {
    x <- x[!is.na(x)]           # remove all NA's
  } else if ( any(is.na(x)) )   # if there is an NA, then return NA
  {
    return(NA)
  }
  n <- lenght(x)
  if ( n == 0 ) { return(NA) }
  half <- (n+1)/2               # e.g. if n=6 then half=3.5 if n=7, then half=4
  if ( n%%2==1 )
  {
    # If n is odd, sort x until the index 'half' is placed correctly
    y <- sort(x, partial = half)[half]
  } else
  {
    # If n is even, sort x until the indices 'half' and 'half+1' are
    # placed correctly. Then return the midpoint of these two elements.
    v <- sort(x,partial= c(half,half+1)) [c(half,half+1)]
    y <- sum( v ) / 2
  }
  return(y)
}
```

Now we execute mymedian() to see what happens. Let the definition of mymedian() be the content of the file 'mymedian.R' in the current directory.

```
> source("mymedian.R")
> mymedian(TRUE)
Error in mymedian(TRUE) : need numeric data
> mymedian(c(4:1,NA,14:11))
[1] NA
> mymedian(c(4:1,NA,14:11),na.rm=TRUE)
[1] 7.5
```

6.6 How to avoid slow R code

When applying your R script to a data set, you might end up waiting a long time until the script finishes. This might even happen for medium size data sets. In that case your R code is presumably slow. Now we explain which code is fast and which code is slow.

Here is some background knowledge. In case of a compiler language, the program code is translated into machine readable code ('compiled') which produces an executable file. This executable file contains only commands which the processor directly understands. R, however, is an interpreter language. Here your program code is not translated into machine code by a compiler. Instead the code is 'interpreted' by the interpreter R at run-time. So your code is translated into machine commands each time you run your script. Obviously this results in a slower run-time which is a disadvantage of interpreter languages. The advantage of R language is that the type (numeric vector, integer matrix) of a variable is not needed to be specified in the script; it is determined at run-time. This is convenient for the programmer.

Here is how you avoid slow R code.

- When using a loop, always initialize new objects to their full final size before the loop instead of growing them at each iteration.
- Try to apply the function to a whole vector instead of iterating over its comnponents (like using sum().
- When it is possible use the commands apply(), lapply(), tapply() introduced in Subsection 6.7.
- BUT do not avoid using loops just for the sake of avoiding loops.

6.7 The commands lapply() and tapply()

The command apply() and its relatives apply a function to each element of the specified object. Here are the relatives of apply().

lapply()	for vectors, 'l'ists and data frames. Returns a list
sapply()	same as lapply() but sappl() tries to 's'implify its output
apply()	applies a function to each row or column of a matrix
tapply()	for 't'ables which are grouped according to factors
mapply()	'm'ultivariate version of sapply()

Examples for lapply(), sapply() and apply():

```
> v <- 1:4
> lapply(v,factorial)     # returns list
[[1]]
```

```
[1] 1

[[2]]
[1] 2

[[3]]
[1] 6

[[4]]
[1] 24

> sapply(v,factorial)     # returns vector
[1]  1  2  6 24
> L <- list(0:3, 5:8, -1:2)
> lapply(L,mean)          # mean of each vector in the list
[[1]]
[1] 1.5

[[2]]
[1] 6.5

[[3]]
[1] 0.5

> sapply(L,mean)
[1] 1.5 6.5 0.5
> m <- cbind(0:3, 5:8, -1:2) ; m
     [,1] [,2] [,3]
[1,]    0    5   -1
[2,]    1    6    0
[3,]    2    7    1
[4,]    3    8    2
> apply(m,2,mean)          # apply 'mean' to each column; 2 for second margin
[1] 1.5 6.5 0.5
> apply(m,1,mean)          # apply 'mean' to each row; 1 for first margin
[1] 1.333333 2.333333 3.333333 4.333333
```

The command tapply() is typically applied to data frames. This command is frequently used and therefore important. To motivate its usage we begin with an example. Consider the following data frame. Each individual is either smoker or non-smoker and belongs to one the three weight classes 1, 2 or 3.

```
> riscfactors <- data.frame( weightcls=rep( 3:1,c(4,4,4) ),
+ smoker=rep(c(0,0,1),4), lifespan=seq(50,72,2) )
> riscfactors
   weightcls smoker lifespan
1          3      0       50
2          3      0       52
3          3      1       54
4          3      0       56
5          2      0       58
```

```
6          2     1      60
7          2     0      62
8          2     0      64
9          1     1      66
10         1     0      68
11         1     0      70
12         1     1      72
> attach(riscfactors)
```

What is the influence of the two risk factors 'weight' and 'smoking' on the average life span? For example we could calculate the average lifespan for smokers and non-smokers in our self-generated data. This could be done as follows.

```
> mean( lifespan[ smoker==0 ] )
[1] 60
> mean( lifespan[ smoker==1 ] )
[1] 63
```

However this becomes inconvenient if the factor has many values. More convenient is the command tapply(). The following command applies the function 'mean' to the two subvectors of lifespan which are determined by the vector 'smoker'.

```
> tapply(lifespan,smoker,mean)
 0  1
60 63
> tapply(lifespan,weightcls,mean)  # group lifespan according to weightcls
 1  2  3
69 61 53
```

7 Linear Regression

7.1 Introduction

Linear regression (the term was first used by Pearson, 1908) is the process of finding a straight line that best approximates a set of points. Suppose we have two variables x and y with decimal values (e.g. height, weights, volumes or temperatures). The variable x shall be the explanatory variable. We think of the response variable y as depending on x or as noisy measurement of a function of x. In accordance with the principle of parsimony, we assume this dependence to be the simplest model of all namely the linear model:

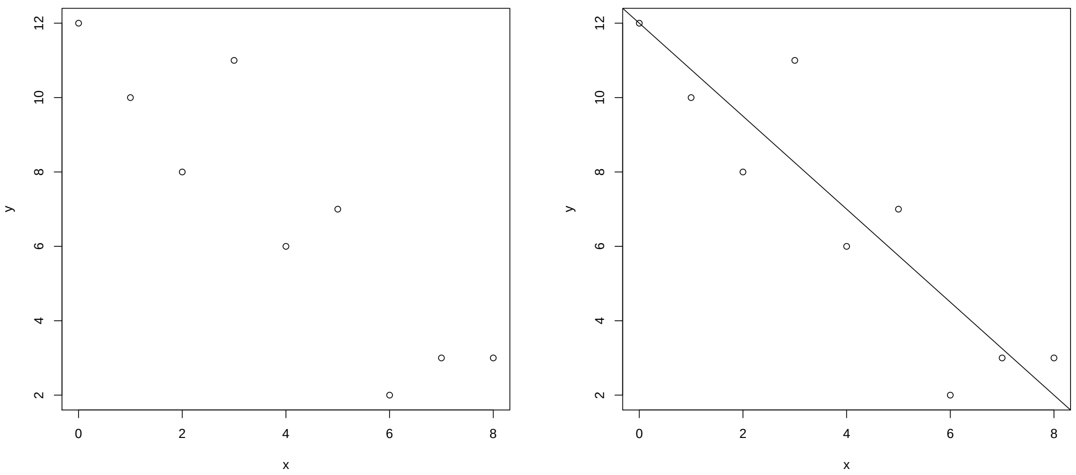
$$y = a + b{\cdot}x$$

The parameter a is called *intercept* (the value of y when $x = 0$) and the parameter b is the *slope.*

Let us consider the following self-generated data. You may think of x as the amount of some growth inhibitor and of y as the measured growth.

```
> x <- 0:8
> y    <- c(12,10,8,11,6,7,2,3,3)
> plot(x,y)
```

The points are depicted in the left of the following two plots.



This is how we do linear regression 'by eye'. We look for the straight line which fits the data best. So let us see what has happened to the response variable y. The values of y decrease from 12 to about 2. At $x = 0$ we have the response variable to be about 12. So the intercept is about $a = 12$. What might be the slope? The values of y decrease from 12 to about 2, so $\Delta y = -10$. The values of x increase from 0 to 8, so $\Delta x = 8$. Altogether we guess the slope to be $\frac{\Delta y}{\Delta x} = \frac{-10}{8} = 1.25$. Let us draw the line given by the linear function

$$y = 12 - 1.25{\cdot}x$$

into the plot of the data points, see the picture on the right-hand side. This looks like a reasonable fit.

Now we ask R for the best fit. We wish to express that 'y is modelled as a function of x'. This is shortly denoted in R as $y \sim x$. We look for the linear model which explains y as a function of x. The corresponding R command is lm($y \sim x$). Note that *lm* is short for *l*inear *m*odel.

```
> lm(y~x)

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
     11.756       -1.217
```

So the straight line which fits the data best is given by

$$y = 11.756 - 1.217{\cdot}x.$$

The object returned by lm() is a very detailed list. One of its entries is 'coefficient' which is a vector of length 2 whose first entry is the intercept and whose second entry is the slope.

```
> regr.obj <- lm(y~x)
> cat("The intercept is",regr.obj$coefficient[1],"and the slope is",
+ regr.obj$coefficient[2],"\n")
The intercept is 11.75556 and the slope is -1.216667
```

The command coef() returns the coefficients as well. For example coef(regr.obj) is the same as regr.obj$coefficient.

7.2 Background

We have not clarified what we mean by 'best fit'. The 'best fit' linear model is found by minimising the error sum of squares. More formally, (intercept, slope) are the values of $(\tilde{a}, \tilde{b})$ which minimise

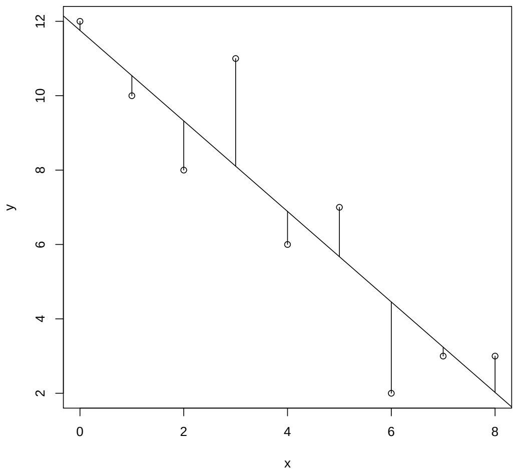
$$\sum_{i=1}^{\text{length}(x)} \big(y_i - \tilde{a} - \tilde{b}{\cdot}x_i\big)^2.$$

Now we need to know whether the linear model is a good explanation of the data. The linear model is supposed to explain the variance or more precisely the sum of squares

$$\text{SSY} <\!- \;\; (n-1) * \text{var}(y) = \sum_{i=1}^{\text{length}(y)} (y_i - \text{mean}(y))^2 = \sum_{i=1}^{\text{length}(y)} (y_i)^2 - \frac{(\sum y_i)^2}{n}$$

of the vector y. Of course the linear model typically does not explain all of SSY, there remain errors. The difference between each data point and the value predicted by the model at the same value of x is called a **residual**. The following figure depicts the residuals in our data. For this we use the R command predict() which calculates the values of the regression line at the points of x.

```
> fitted <- predict(regr.obj)
> plot(x,y)
> abline(regr.obj)
> for( i in 1:9 ){ lines( c(x[i],x[i]),c(y[i],fitted[i]) ) }
```



The linear model does not explain the sum of the squares of the residuals

$$\text{SSE} <\!- \sum_{i=1}^{\text{length}(x)} \big(y_i - a - b{\cdot}x_i\big)^2$$

where a is the intercept and b is the slope of the regression line. The explained variation is the regression sum of squares

$$\text{SSR} \;<\!- \;\; \text{SSY} - \text{SSE}\,.$$

One can show that

$$\mathrm{SSR} = \frac{\big(\mathrm{SSXY}\big)^2}{\mathrm{SSX}} \quad \text{where } \mathrm{SSXY} = (n-1)\cdot\operatorname{cov}(x,y), \quad \mathrm{SSX} = (n-1)\cdot\operatorname{var}(x).$$

A measure of fit is the fraction of the total variation in y that is explained by the regression. The total variation is SSY and the explained variation is SSR, so our measure of fit – let's call it r^2 – is given by

$$r^2 = \frac{\mathrm{SSR}}{\mathrm{SSY}} = \big(\operatorname{cor}(x,y)\big)^2.$$

The formal name of this quantity is the coefficient of determination, but most people just refer to it as '*r squared*'. The value of r^2 lies between 0 and 1. The bigger r^2, the better is the fit. If the fit is perfect, then $r^2 = 1$. At the other extreme if x explains no variation in y at all, then $r^2 = 0$.

```
> n <- length(x)
> SSY <- (n-1)*var(y) ; SSY
[1] 108.8889
> SSR <- ( (n-1)*cov(x,y) )^2/ ( (n-1)*var(x) ) ; SSR
[1] 88.81667
> SSE <- SSY-SSR ; SSE
[1] 20.07222
> SSR/SSY ; ( cor(x,y) )^2     # r^2
[1] 0.8156633
[1] 0.8156633
```

These values are now recorded in what is known as 'Anova table'. Anova is short for analysis of variance.

Source	Sum of squares	Degrees of freedom	Mean squares	F value	Pr(>F)
Regression	88.817				
Error	20.072				
Total	108.889				

The third column is important to understand. There are $n <-$ length(x) points in the graph. As these points are sampled independently from the underlying distribution, there are n degrees of freedom in the data vector y. The vector x is now considered to be fixed and therefore contributes no degrees of freedom. In the total sum of squares $\mathrm{SSY} = \sum(y - \mathrm{mean}(y))^2$, the vector y is centred with its mean. The resulting vector has mean 0. So the degree of freedom of the centred vector is now $n-1$ because knowing $n-1$ element of a centred vector enables us to calculate the n-th element. Consequently the degree of freedom in the total sum of squares is $n-1$ (in our case 8). A different argument for this is that there is one parameter estimated from y namely the mean mean(y). Hence there remain $n-1$ degrees of freedom. Next consider the error sum of squares $\mathrm{SSE} = \sum(y - a - b\cdot x)^2$. Here there are two parameters estimated from y, namely a and b. Hence there remain $n-2$ degrees of freedom (in our case 7). Because of $\mathrm{SSR} = \mathrm{SSY} - \mathrm{SSE}$ there remains $n-1-(n-2) = 1$ degree of freedom for the regression sum of squares. Another way to see this is from $\mathrm{SSR} = \mathrm{SSXY}^2 \,/\, \mathrm{SSX}$ which contains exactly one parameter which is estimated from y namely $\operatorname{cov}(x,y)$.

To complete the Anova table, we enter the variances in the fourth column. Recall that

$$\text{variance} = \frac{\text{sum of squares}}{\text{degree of freedom}}.$$

The fifth column contains the F ratios. In most simple Anova tables, you divide the treatment variance in the numerator (here the regression variance) by the error variance in the denominator. The last column contains the probability to obtain the F ratio or a higher value by chance. The distribution of the ratio of the mean squares is the F-distribution, see Subsection 2.1. So we obtain the probability of having the F ratio or a higher value with the command `1-pf()`.

```
> 20.072/7
[1] 2.867429
> 108.889/8
[1] 13.61112
> 88.817/2.86746
[1] 30.9741
> 1-pf(30.9741,1,7)
[1] 0.0008460645     # the regression line is highly significant
```

Source	Sum of squares	Degrees of freedom	Mean squares	F value	Pr(>F)
Regression	88.817	1	88.817	30.974	0.000846
Error	20.072	7	2.867		
Total	108.889	8	13.611		

7.3 Summary.aov() and summary()

R produces the above Anova table without the last line with the command summary.aov() which summarises an analysis of variance model.

```
> summary.aov(regr.obj)
            Df Sum Sq Mean Sq F value   Pr(>F)
x            1 88.817  88.817  30.974 0.000846 ***
Residuals    7 20.072   2.867
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

The command summary() shows more details and contains everything you need to know about the parameters and their standard errors.

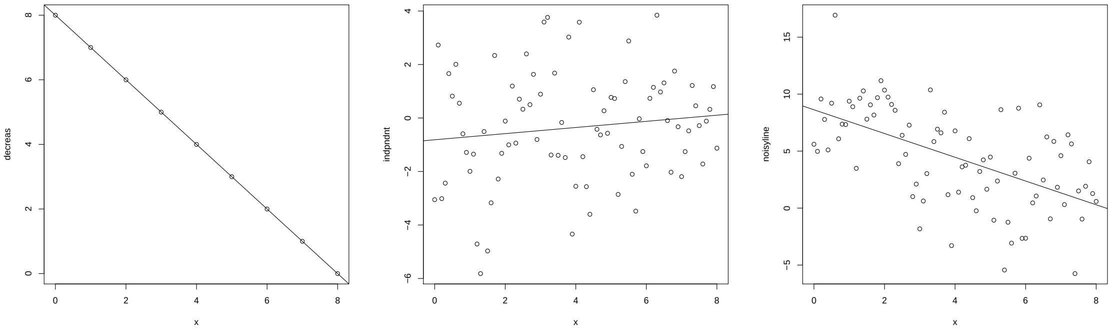
```
> summary(regr.obj)

Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-2.4556 -0.8889 -0.2389  0.9778  2.8944

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  11.7556     1.0408  11.295 9.54e-06 ***
x            -1.2167     0.2186  -5.565 0.000846 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

```
  Residual standard error: 1.693 on 7 degrees of freedom
  Multiple R-squared: 0.8157,djusted R-squared: 0.7893
  F-statistic: 30.97 on 1 and 7 DF,  p-value: 0.000846
```

We read off from the summary that both the estimate of the intercept and the estimate of the slope are highly significant (at level 0.001).

Here are two extreme examples and one moderate example with self-generated data.

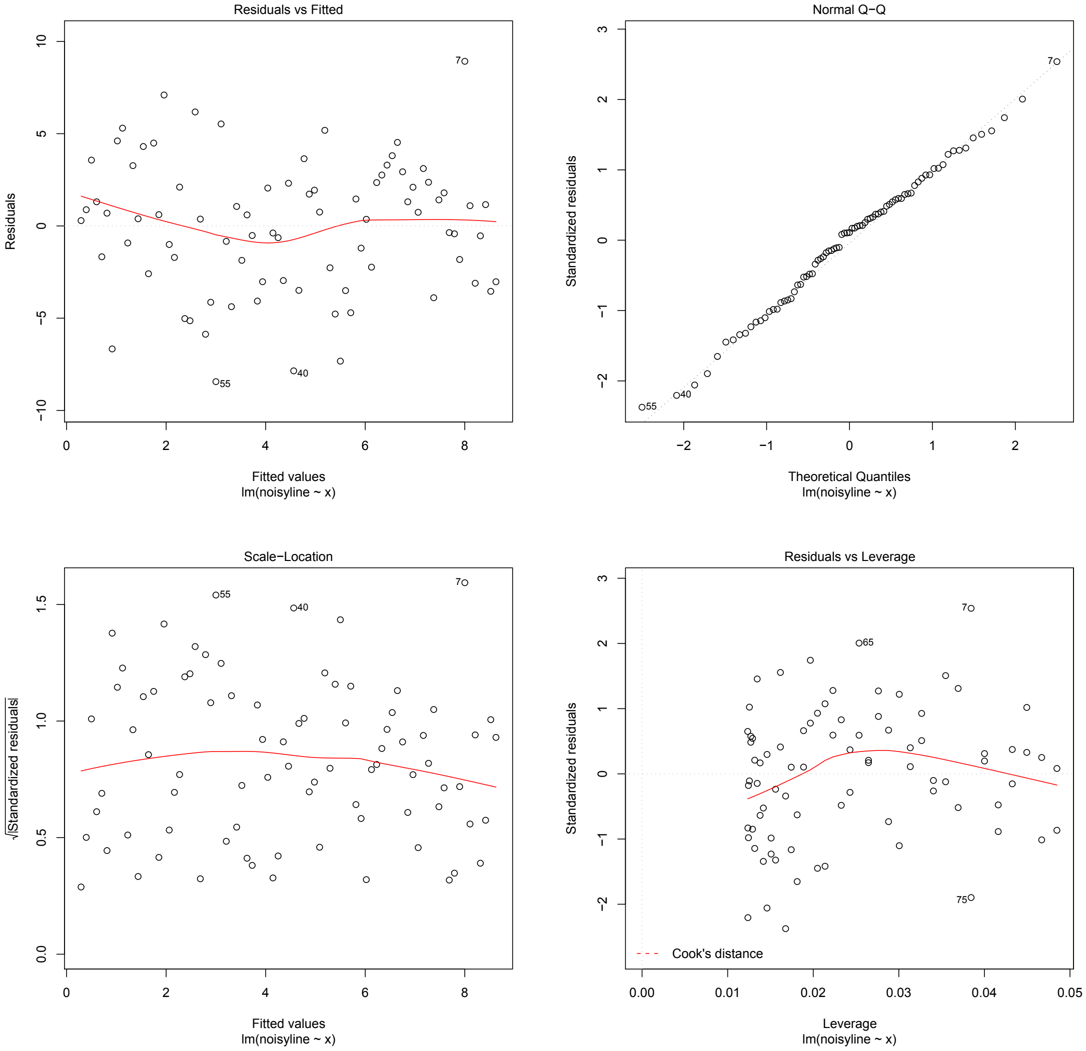
```
# the first example is a decreasing line with slope -1 without noise.
  > x <- 0:8
  > decrease <- 8:0
  > regr <- lm(decrease˜x)
  > regr1 <- lm(decrease˜x)
  > plot(x,decrease) ; abline(regr1)
  > summary(regr1)

  Call:
  lm(formula = decrease ˜ x)

  Residuals:
         Min         1Q     Median         3Q        Max
  -1.724e-15  1.370e-16  2.031e-16  2.472e-16  9.006e-16

  Coefficients:
                Estimate Std. Error    t value Pr(>|t|)
  (Intercept)  8.000e+00  4.894e-16  1.635e+16   <2e-16 ***
  x           -1.000e+00  1.028e-16 -9.729e+15   <2e-16 ***
  ---
  Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

  Residual standard error: 7.962e-16 on 7 degrees of freedom
  Multiple R-squared:     1,djusted R-squared:     1
  F-statistic: 9.465e+31 on 1 and 7 DF,  p-value: < 2.2e-16

# the second example is just noise which does not depend on x
  > x <- seq(0,8,by=0.1)
  > length(x)
  [1] 81
  > indpndnt <- rnorm(81,0,2)
  > regr2 <- lm(indpndnt˜x)
  > plot(x,indpndnt) ; abline(regr2)
  > summary(regr2)

  Call:
  lm(formula = indpndnt ˜ x)

  Residuals:
      Min      1Q  Median      3Q     Max
  -5.1527 -1.2588 -0.1359  1.2855  4.2083
  Coefficients:
              Estimate Std. Error t value Pr(>|t|)
  (Intercept) -0.81396    0.45566  -1.786   0.0779 .
  x            0.11480    0.09835   1.167   0.2466
  ---
  Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

  Residual standard error: 2.07 on 79 degrees of freedom
  Multiple R-squared: 0.01695,djusted R-squared: 0.004511
  F-statistic: 1.362 on 1 and 79 DF,  p-value: 0.2466

# the third example is a decreasing line with slope -1/2 plus noise
  > noisyline <- seq(8,0,by=-0.1) + rnorm(81,0,4)
  > regr3 <- lm(noisyline~x)
  > plot(x,noisyline) ; abline(regr3)
  > summary(regr3)

  Call:
  lm(formula = noisyline ~ x)

  Residuals:
      Min      1Q  Median      3Q     Max
  -8.4352 -2.5893  0.3918  2.3173  8.9245

  Coefficients:
              Estimate Std. Error t value Pr(>|t|)
  (Intercept)   8.6273     0.7892  10.932  < 2e-16 ***
  x            -1.0420     0.1703  -6.117 3.43e-08 ***
  ---
  Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

  Residual standard error: 3.584 on 79 degrees of freedom
  Multiple R-squared: 0.3214,djusted R-squared: 0.3129
  F-statistic: 37.42 on 1 and 79 DF,  p-value: 3.429e-08
```



7.4 Model checking

The p-values are one thing to check. In addition one should check for **constancy of variance** and **normality of errors**. The simplest way to do this is with four model-checking plots:

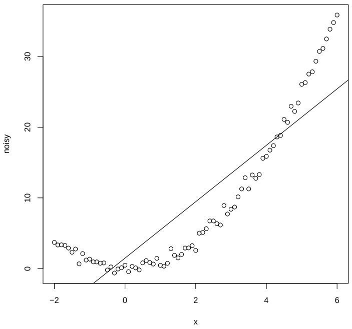
```
> plot(regr3)
```



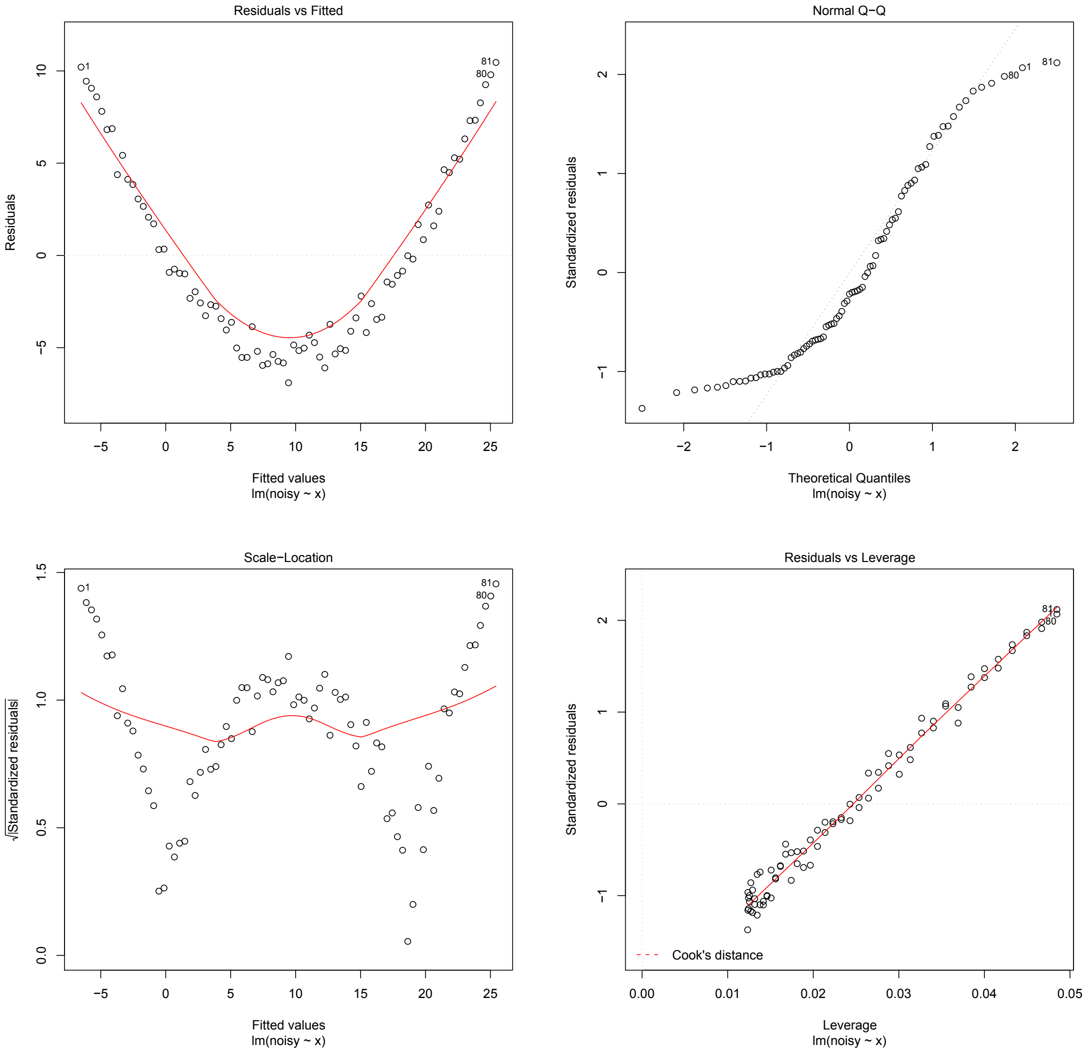
This is how it should look like: The points of the upper left picture looks like points from a centred normal distribution and the line is identically zero. The points in the upper right picture should be close to the dotted line. We do not discuss the third and fourth picture.

Here is an example in which the regression line fits poorly.

```
> x <- seq(-2,6,by=0.1)
> noisy <- x^2 + rnorm(81,0,0.1)
> regr4 <- lm(noisy~x)
> plot(x,noisy) ; abline(regr4)
```


noisy
0
10
20
30
−2
0
2
4
6
x

```
> plot(regr4)
```


Residuals vs Fitted
Residuals
−5
0
5
10
Fitted values
lm(noisy ~ x)
1
81
80
Normal Q−Q
Standardized residuals
−1
0
1
2
−2
−1
0
1
2
Theoretical Quantiles
lm(noisy ~ x)
1
80
81
Scale−Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
−5
0
5
10
15
20
25
Fitted values
lm(noisy ~ x)
1
81
80
Residuals vs Leverage
Standardized residuals
−1
0
1
2
0.00
0.01
0.02
0.03
0.04
0.05
Leverage
lm(noisy ~ x)
Cook's distance
81
80

8 Advanced topics

8.1 Generating and manipulating strings

Here is an overview of commands which generate or manipulate strings. For the syntax details, see the respective help pages.

`format()`	Format an R object for pretty printing
`formatC()sprintf()`	Formatting as in the language 'C'
`grep()grepl()`	
`regexpr()`	
`gregexpr()`	Find a substring in a list of strings (*g*et *r*egular *exp*ression)
`match()pmatch()`	
`charmatch()`	Returns positions of (partial) matches
`nchar()`	Count the *n*umber of *char*acters
`parse()`	Convert a string into an expression
`deparse()`	Convert an expression into a string
`paste()`	Concatenate strings after converting to string
`cat()`	Same as paste() but prints result onto the console (or into a file)
`strsplit()`	*Split str*ing at given delimiter
`sub()gsub()`	*Sub*stitute a substring by another string
`substring()`	Return the substring between the given positions
`toupper()`	Translate lower-case into upper-case characters
`tolower()`	Translate upper-case into lower-case characters

Examples

```
> s <- paste("Diet",1:8); s # adds one space between the arguments by default
[1] "Diet 1" "Diet 2" "Diet 3" "Diet 4" "Diet 5" "Diet 6" "Diet 7" "Diet 8"
> s <- paste("grey",2:10,"0",sep=""); s
[1] "grey20"  "grey30"  "grey40"  "grey50"  "grey60"  "grey70"  "grey80"
[8] "grey90"  "grey100"
> nchar("Number?")
[1] 7
> nchar(s)
[1] 6 6 6 6 6 6 6 6 7
> u <- toupper(s) ; u
[1] "GREY20"  "GREY30"  "GREY40"  "GREY50"  "GREY60"  "GREY70"  "GREY80"
[8] "GREY90"  "GREY100"
> tolower(u)
[1] "grey20"  "grey30"  "grey40"  "grey50"  "grey60"  "grey70"  "grey80"
[8] "grey90"  "grey100"
> substring("abcdef",first=2,last=4)
[1] "bcd"
> s[2:4] <- "green"
> substring(s,rep(5,9),rep(10,9)) # characters between position 5 and 10
[1] "20"  "n"   "n"   "n"   "60"  "70"  "80"  "90"  "100"
```

```
> strsplit("Peter Pan"," ")
[[1]]
[1] "Peter" "Pan"

> strsplit("Peter Pan","e")
[[1]]
[1] "P"     "t"     "r Pan"
```

Here is an application of `parse()` and one of `deparse()`.

```
> c <- "green"
> str <- paste(’plot(1:10,col="’,c,’")’,sep="") ; str
[1] "plot(1:10,col=\"green\")"
> ep <- parse(text=str)
> eval(ep)              # Execute the command
> x <- 1:10
> regr <- lm(x˜1)
> epp <- formula(regr)  # extract the formula of the call to lm() from regr
> class(epp)
[1] "formula"
> deparse(epp)          # convert the formula into a string
[1] "x ˜ 1"
> pmatch(c("mod","med"), c("mean", "median", "mode"))
[1] 3 2                 # third element begins with ’mod’, second with ’med’
> format(13.7)
[1] "13.7"
> format(13.7, nsmall = 3)
[1] "13.700"
> format(c(6.0, 13.1), digits = 2)
[1] " 6" "13"
> format(c(6.0, 13.1), digits = 2, nsmall = 1)
[1] " 6.0" "13.1"
```

A quite strong tool for dealing with strings are regular expressions. Here is a first simple example. The command `grep()` finds its first argument in a list of strings and returns the index vector (default) or the strings (if `value=TRUE`).

```
> grep("rey",s)
[1] 1 2 3 4 5 6 7 8 9
> grep("rey",s,value=T)
[1] "grey20"  "grey30"  "grey40"  "grey50"  "grey60"  "grey70"  "grey80"
[8] "grey90"  "grey100"
```

Here a fixed string is found in a list of strings. The power of regular expressions is that not only fixed strings but quite general strings can be matched. For this the following placeholders and quantifiers are used.

.	matches any single character
[abd]	matches any single character in the list, here a b or d
[ˆabd]	matches any single character except the characters in the list
[b-s]	matches any single character between b and s
*	the preceding item will be matched zero or more times
+	the preceding item will be matched one or more times
?	the preceding item will be matched zero or one time
{n,}	the preceding item will be matched n or more times
{n,m}	the preceding item will be matched at least n times but not more than m times
abc|def	matches one or the other string, here it matches 'abs' or 'def'
ˆ	matches the beginning of the string
$	matches the end of the string
()	the string which is matched between (and) can later be referred to with \\1, \\2 etc

There are more features, see `?regexp` or `?grep`.

By default repetition is greedy, so the maximal possible number of repeats is used. This can be changed to minimal by appending ? to the quantifier.

Here are several examples to become familiar with regular expressions.

```
> s <- colors()
> grep("gr[ea]y",s)          # any color with 'grey' or 'gray' in it
> grep("gr[ea]y",s,value=T)
> grep("gr[ea]y$",s,value=T) # any color which ends on 'gr[ea]y'
> grep("^gre",s,value=T)     # any color which starts with 'gre'
> grep("^gre.*[^0-9]$",s,value=T) # starts with 'gre' and ends with a letter
> grep("green|yellow",s,value=T)  # any color with 'green' or 'yellow' in it
> sub("AB","ab","ABCDEFAB")
[1] "abCDEFAB"
> gsub("AB","ab","ABCDEFAB")
[1] "abCDEFab"
> str <- c("data1 ","data2   ","data3  ")
> sub(" +$","",str)                          # trim trailing white space
[1] "data1" "data2" "data3"
> gsub("([ab])", "\\1_\\1_", "abc and ABC") # double all 'a' or 'b's
[1] "a_a_b_b_c a_a_nd ABC"
> gsub("\\b(\\w)", "\\U\\1", "a test of capitalizing", perl=TRUE)
[1] "A Test Of Capitalizing"
```

8.2 Object-oriented programming

There are two approaches to object-oriented programming in R. Both have advantages and disadvantages. We first explain the so called S3 approach which was historically the first approach.

Many objects in R have a 'class' attribute, a character string or character vector giving the name(s) of the class(es) the object belongs to. If the object does not have a class attribute, then it has an implicit class which is the result of `mode(x)` (except that integer vectors have implicit class 'integer'). You obtain the class name(s) of an object `obj` with the command `class(obj)`. The class attribute can be set with `class(obj) <-`. If an object has a 'class' attribute, then `is.object()` returns `TRUE`.

The most useful feature of object-oriented programming are 'generic' functions. For example, `print()` is a generic function. If `regr` is of class `lm`, `print(regr)` calls automatically the function `print.lm(regr)`. If 'clname' is a class and the function `print.clname` exists, then `print()` calls this function `print.clname` if its argument is of that class. More generally if 'genericfun' is then name of generic function, if `obj` is of class 'clname' and if the function `genericfun.clname` exists, then `genericfun(obj)` automatically calls `genericfun.clname(obj)`. If the function `genericfun.clname` does not exist, then the default method `genericfun.default(obj)` is called. The command `print()` generic function with most methods, see `methods("print")`.

Here is an educational example to illustrate the mechanism.

```
> Q <- 1              # arbitrary object
> class(Q) <- "quit"  # Q is now of class 'quit' (which didn't exist before)
> print.quit <- function(x) q("no")
> Q                   # closes R session without saving workspace image
```

Entering `Q` calls implicitly `print(Q)`. As `Q` is of class 'quit', this calls automatically `print.quit(Q)`. We defined this function to execute `q("no")` which quits the R session without saving a workspace image. This example is of no practical use but it illustrates the mechanism of generic functions.

Here is a list of commands for object-oriented programming:

`attributes()`	Returns the list of all attributes of an object
`attr()`	Viewing and setting single attributes
`class()`	Viewing and setting the class attribute
`getS3method()`	Shows an invisible method
`inherits()`	Inherit methods from another class
`methods()`	Show all methods belonging to a generic function or to a class
`unclass()`	Removes the 'class' attribute. (default generic function is called)

```
> x <- 1:4; y <- c(1.2,pi)
> class(x)
[1] "integer"
> class(y)
[1] "numeric"
> class(as.factor(x))
[1] "factor"
> class(list(x,y))
[1] "list"
> regr <- lm(x^2~x)
> class(regr)
[1] "lm"
> fun <- ecdf(rnorm(100))
> class(fun)
[1] "ecdf"     "stepfun"  "function"
> attributes(regr)
$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
```

```
$class
[1] "lm"

> print
function (x, ...)
UseMethod("print")
<environment: namespace:base>
> methods(print)
  [1] print.acf*
  ...
 [42] print.data.frame
 ...
 [44] print.default
 ...
 [56] print.factor
 ...
 [61] print.glm
 ...
 [75] print.lm
 ...
[101] print.quit            # the function which we defined above
 ...
[143] print.xtabs*

   Non-visible functions are asterisked
> methods(class=factor)     # all methods of class 'factor'
 [1] [.factor              [[.factor             [<-.factor
 [4] all.equal.factor      as.character.factor  as.data.frame.factor
 [7] as.Date.factor        as.list.factor        as.POSIXlt.factor
[10] as.vector.factor      format.factor         is.na<-.factor
[13] length<-.factor       levels<-.factor       Math.factor
[16] Ops.factor            plot.factor*          print.factor
[19] relevel.factor*       relist.factor*        reorder.factor*
[22] rep.factor            summary.factor        Summary.factor
[25] xtfrm.factor

   Non-visible functions are asterisked
> line        # the function 'line' returns an object of class 'tukeyline'
function (x, y = NULL)
{
  xy <- xy.coords(x, y)
  ok <- complete.cases(xy$x, xy$y)
  n <- length(ok)
  if (n <= 1)
      stop("insufficient observations")
  z <- .C("tukeyline", as.double(xy$x[ok]), as.double(xy$y[ok]),
      double(n), double(n), n, double(2), DUP = FALSE, PACKAGE = "stats")
  value <- list(call = sys.call(), coefficients = z[[6]], residuals = z[[3L]],
      fitted.values = z[[4L]])
```

```
    class(value) <- "tukeyline"
    value
  }
```

For example the command `print.factor()` is the print method for class 'factor'.

As educational example let us define a class 'myline'. The class is represented by a list consisting of an 'intercept' and a 'slope'. Here is an example how to generate an object of this class.

```
> obj <- list(intercept=5,slope=2)
> class(obj) <- "myline"
> obj        # print.myline is not defined, so print.default is called
$intercept
[1] 5

$slope
[1] 2

attr(,"class")
[1] "myline"
```

Next we define a print method.

```
> print.myline <- function(x,...){
+ cat("Intercept:\t",x$intercept,"\nSlope:    \t",x$slope,"\n")}
> obj
Intercept:    5
Slope:        2
```

So far `plot(obj)` goes pretty wrong, so let's define a plot function.

```
> plot.myline <- function(x,...){
+ x<- seq(-5,5,by=0.1)
+ plot(x,obj$intercept+obj$slope*x,type="l",ylab=NA,xlab=NA,...)
+ }
> plot(obj,sub="mysub")
```

Next one should suitably define other functions such as `summary.myline()` or a function which generates objects of class 'myline'.

Note that the S3 approach has no strict definition of the class itself. For example we could generate an object of class 'myline' which includes more information:

```
> obj2 <- list(intercept=5,slope=2,point=1)
> class(obj2) <- "myline"
> obj2          # calls print.myline
Intercept:   5
Slope:       2
```

Let us now introduce the S4 approach to object-oriented programming. Note that the following commands only work on S4 objects. The idea is to provide a command `SetClass()` which explicitly defines a class including all of its components. Here we only give an example to illustrate the main idea. Note the following differences: Entering an S4 object on the console calls `show()` instead of `print()`. Moreover elements of an S4 object are accessed with the @-operator rather than with the $-operator. The elements of an S4 objects are called `slots`. Another way of accessing elements is with the command `slot(`*obj,slotname*`)`.

Let us define the class 'myline' in the S4 approach. The command `setClass()` defines a new class. The command `getClass()` returns the definition of the class. The command `prototype()` defines the default object of class 'myline'.

```
> setClass("myline",
+     representation=representation(intercept="numeric",slope="numeric"),
+     prototype     =prototype(intercept=0,slope=1)
+ )
[1] "myline"
> getClass("myline")
Class myline [in ".GlobalEnv"]

Slots:

Name:  intercept     slope
Class:   numeric   numeric

> getSlots("myline")   # get the slots of class 'myline'
intercept     slope
"numeric" "numeric"
> slotNames("myline")  # same as names(getSlots("myline"))
[1] "intercept" "slope"
```

The command `new()` generates a new object of a class.

```
> obj <- new("myline")        # Generate a new object of class 'myline'
> obj                         # This implicitly calls show(obj)
An object of class myline
Slot "intercept":
[1] 0

Slot "slope":
[1] 1
```

Next we define the method `show()` with `setMethod()`

```
> setMethod("show", signature(object="myline"),
+    function(object){
+      cat("Intercept:\t",object@intercept,"\nSlope:    \t",object@slope,"\n")
+    }
+ )
[1] "show"
> obj                          # implicitly calls the method 'show()'
Intercept:   0
Slope:       1
> slotNames(obj)
[1] "intercept" "slope"
> obj@slope
```

```
[1] 1
> obj@slope <- 5
> slot(obj,"intercept")        # same as obj@intercept
[1] 0
> slot(obj,"intercept") <- 3
> obj
Intercept:   3
Slope:       5
> scndobj <- new("myline",intercept=6,slope=8)
> scndobj
Intercept:   6
Slope:       8
> unclass(scndobj)
<S4 Type Object>
attr(,"intercept")
[1] 6
attr(,"slope")
[1] 8
```

Advantage of S3 approach: Quick and easy to implement. Advantage of S4 approach: Has a formal class definition and enforces to think about the class structure. This avoids later problems.

Knowing how classes and objects are defined in R we can investigate objects which are returned by more complex commands.

```
> x <- 1:10
> regr <- lm(x~1)
> class(regr)
> methods(class="lm")
> attributes("lm")
> attributes(regr)
> unclass(regr)
> tt <- t.test(rnorm(100))
> class(tt)
> methods(class="htest")
> attributes(tt)
> unclass(tt)
> fun <- ecdf(rnorm(100))
> class(fun)
> methods(class="ecdf")
> attributes(fun)
> unclass(fun)
```

8.3 Scoping rules

R looks for variables in the search path which is an hierarchical list. The workspace `.GlobalEnv` is element 0 in that list. All elements being defined on the console or in a script (outside of a function) are stored in this workspace. Other elements of the search path are loaded libraries, attached data frames or environments generated by a function call. The command `search()` returns the list of elements with non-positive levels. At the startup of an R session the search path could look as follows.

```
> search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

Ignore the enumeration here. The workspace is element 0, `package:stats` is element -1, the element on level -2 is `package:graphics` and `package:base` is element -8 in this case. If a data frame is attached or a library is loaded, then these are placed at position -1. The contents of each element of the search path is viewed with `ls()`.

```
> attach(ChickWeight)
> library("sfsmisc")
> attach(cars)
> search()
 [1] ".GlobalEnv"        "cars"              "package:sfsmisc"
 [4] "ChickWeight"       "package:stats"     "package:graphics"
 [7] "package:grDevices" "package:utils"     "package:datasets"
[10] "package:methods"   "Autoloads"         "package:base"
> x <- 5
> ls()           # returns the workspace by default
[1] "x"
> ls("cars")
[1] "dist"  "speed"
> detach(ChickWeight)
> detach(cars)
> detach(package:sfsmisc)
```

The ordering of the search path has the following reason. If there are different objects of the same name on different levels, then R uses the object on the highest level. For example, the library `base` contains the function `sum()`. If you define `sum` on the console, then this object is stored in `.GlobalEnv` which is at a higher level. The originial function is thereby 'hidden'. You can still access the original function through the `::`-operator as `base::sum`. An existing object is removed from the search path with the command `rm()`.

```
> sum(1:5)
[1] 15
> sum <- exp      # define a new object called 'sum' in the workspace
> sum(1:5)
[1]   2.718282   7.389056  20.085537  54.598150 148.413159
> base::sum(1:5)  # the originial function is accessed with base::sum
[1] 15
> rm(sum)         # Remove the object 'sum' from the workspace
> sum(1:5)        # Now base::sum() is no longer hidden
[1] 15
```

If you call a function from the console, then this call creates a new element in the search path at level 1. This element contains all objects which are created within the function. When the function terminates, then the returned value is copied into the workspace and the environment at level 1 in the search path is deleted. If another function is called from within the function, then an environment is created at level 2 and so on. This hierarchical search path ensures that R uses the object which has been defined latest. Consider the following example:

```
> x <- 5
> myfun <- function(do.it=TRUE){
+   if(do.it) x <- 3
+   innerfun <- function() print(x)
+   innerfun()
+ }
> myfun()
[1] 3
> myfun(FALSE)
[1] 5
```

The workspace contains a variable x. The call of myfun() creates a new environment at position 1 in the search path. If myfun() is called with do.it=TRUE, then a variable x is added to level 1. The call of innerfun() then creates an environment at level 2. At the command print(x), R looks for a variable x and finds none on level 2 but finds one on level 1 which has the value 3. So print(x) prints 3. If myfun() is called with do.it=FALSE then no variable x is added to level 1. The call of innerfun() then creates an environment at level 2. At the command print(x), R looks for a variable x and finds none on level 2 or 1 but finds one on level 0 (the workspace) which has the value 5. So print(x) prints 5.

A programming style as in myfun() can lead to surprising results as the output depends on the current configuration of the search path. It is therefore considered as 'bad style'. It is recommended to always pass variables as arguments. Thereby the code is easier to read and mistakes are avoided.

8.4 Regular expressions

The command scan() reads numeric data from a file. The command readLine() reads lines from a file. The command writeLine() writes the elements of a character vector as lines into a file.

```
> cat("TITLE extra line", "2 3 5 7", "11 13 17", file="ex.txt", sep="\n")
> pp <- scan("ex.txt")
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings,  :
  scan() expected 'a real', got 'TITLE'
> pp <- scan("ex.txt", skip = 1, quiet= TRUE)
> cat("TITLE extra line", "2 3 5.34 7", "11 13 17", file="ex2.txt", sep="\n")
> scan("ex2.txt", skip = 1, quiet= TRUE)
[1]  2.00  3.00  5.34  7.00 11.00 13.00 17.00
> scan("ex.txt", skip = 1)
Read 7 items
[1]  2  3  5  7 11 13 17
> scan("ex.txt", skip = 1, nlines=1)        # only 1 line after the skipped one
Read 4 items
[1] 2 3 5 7
> v <- readLines("ex.txt")
> v
[1] "TITLE extra line" "2 3 5 7"          "11 13 17"
> writeLines(v,"exout.txt")
> unlink(c("ex.txt","ex2.txt","exout.txt")) # deletes the file
> sequences <- readLines("sequences.txt")
> sequences
> sequences<-sub(" *\$","",sequences)       # remove trailing spaces
> sequences
> sequences<-gsub("U","T",sequences)        # substitute all U by T

> regexpr("TAG",sequences)                  # find first occurrence
[1]  5  2 31 50 15 -1
attr(,"match.length")
[1]  3  3  3  3  3 -1
> gregexpr("TAG",sequences)                 # find all occurrences
> regexpr("TAG.*TAG",sequences)
[1]  5  2 31 50 15 -1
attr(,"match.length")
[1] 483 579 440 602 917  -1
> sequences
```

8.5 Outlook: DNA and protein data

A fasta-file has a specific format to store sequence data. The command read.fasta() from the package seqinr reads fasta-files. The command translate() from the package seqinr translates nucleic acid sequences into the corresponding peptide sequence.

```
> install.packages("seqinr")               # install package
> library(seqinr)
> nucleo <- read.fasta("norovirus.fas")
> nucleo
> class(nucleo)
[1] "list"
> peptide <- lapply(nucleo,translate)
> peptide

> unlist(sapply(peptide,paste,collapse=""))
> myalign <- t(matrix(unlist(peptide),nrow=189))
>
> allequal <- function(x) {
>   sum(x[1]!=x[2:length(x)])==0
> }
>
> countunequal <- function(x) {
>   sum(x[1]!=x[2:length(x)])
> }
>
> hasX <- function(x) {
>   sum("X"==x[1:length(x)])>0
> }
>
> myalign[,!apply(myalign,2,allequal) & !apply(myalign,2,hasX)]
> names(nucleo)
> myalign[12:19,!apply(myalign,2,allequal) & !apply(myalign,2,hasX)]
> myalign[4:11,!apply(myalign,2,allequal) & !apply(myalign,2,hasX)]
> myalign[,apply(myalign,2,countunequal)>1 & !apply(myalign,2,hasX)]
> library(help="seqinr")          # all commands in seqinr
```

Appendix

A Save and load workspace and command history

If you close your R session, then all variables and objects are deleted unless you answer ”Save workspace image?” with ”yes”. You can store all variables with the command `save.image()` and load them in the next R session with `load()`:

```
> save.image("workspace42.Rdata") # store workspace in current working directory
> # next session:
> load("workspace42.Rdata") # only works if current working directory is correct
> getwd()
[1] "C:/"
> setwd("C:/R")                    # change working directory if necessary
> save.image("C:/R/workspace42.Rdata")# store workspace in specified directory
> # next session:
> load("C:/R/workspace42.Rdata")  # works always (if file exists)
```

Also the commands you typed in could be gone in the next R session. Save the history of your commands with `savehistory()`:

```
> savehistory("commands42.Rhistory")
       # store command history in current working directory
> # next session:
> loadhistory("commands42.Rhistory")
       # only works if current working directory is correct
```

B Customizing R

If you wish to have self-written commands available at the startup of R, then you might want to customize R as follows, see ?Startup for more details. If there exists a file ”.Rprofile” in the current directory or in the user’s home directory, then it is sourced at startup of every R session, that is,

```
> source("./.Rprofile")
```

and the same command with ”./” replaced by user’s home directory are executed automatically at the beginning of every R session.

There are two special functions which are often defined in ’.Rprofile’. These two functions are ’.First’ and ’.Last’ and are executed at the beginning and at the end of an R session, respectively. Moreover you might want to set options for the R session with the command options() in ’.Rprofile’. Here is an example for ’.Rprofile’:

```
.First <- function() { cat("\n  Welcome to R!\n\n") }
.Last <- function() { cat("\n  Goodbye!\n\n") }
set.seed(1234)         # set the seed of the random number generator
options(digits=5)      # see ?print
options(error=recover) # Call recover() whenever an error occurs
```

In addition R searches for ’.Renviron’ in the current working directory and in the user’s home directory. The environment file contains lines in the form ’name=value’ and used to set environment variables. For example if ’.Renviron’ consists of the following line

```
LANGUAGE=en
```

then the language of the R session is set to English so that all warnings and all error messages are printed in English.

C Debugging

The simplest debugging method is to print text and variables in order to see where the error occured and in what state the variables where in. However, this simple debugging method is not always successful.

R provides the following debugging tools.

`browser()`	Starts the browser
`debug(),undebug()`	A function is executed in the browser
`dump.frames()`	Save all variable values at error time
`debugger()`	View values of variables at error time
`traceback()`	Show in which function the error occured
`trace(),untrace()`	Insert debugging code during run-time

The commands 'c', 'n', 'where' and 'Q' are used within the browser in addition to all regular R commands. Here is an example.

```
> fun2 <- function(x, s) {
+   return( x[[s]] + 5 )
+ }
> fun1 <- function(x) {
+   y <- x+1
+   return( fun2(y,s=-5) )
+ }
> fun1(1:5)
Error in x[[s]] : attempt to select more than one element
> traceback()
2: fun2(y, s = -2)
1: fun1(1:5)
```

The coammnd `traceback()` shows that the error occured in function `fun2` which has been called from function `fun1`.

After `debug(fun2)`, every call of `fun2` is executed in the browser.

```
> debug(fun2)
> fun1(1:5)
debugging in: fun2(y, s = -2)
debug: {
    return(x[[s]] + 5)
}
attr(,"srcfile")
funerror.R
Browse[1]> ls()        # list all objects
[1] "s" "x"
```

```
Browse[1]> x           # print the value of x
[1] 2 3 4 5 6
Browse[1]> where       # where are we?
where 1: fun2(y, s = -2)
where 2: fun1(1:5)

Browse[1]> n           # execute next command
debug: return(x[[s]] + 5)
Browse[1]> n
Error in x[[s]] : attempt to select more than one element
> undebug(fun2)        # from now on, do not debut 'fun2'
```