R manual

by A.W. van der Vaart
edited by M.A. Jonker
translated from Dutch by D. Dobler

Faculty of Science
Vrije Universiteit Amsterdam

January 19, 2021

This manual is based on a manual for S-plus that has been written by A.W. van der Vaart. The S-plus manual has been modified by M.A. Jonker such that this work is usable for R. She did that by checking whether the S-plus functions are still existing and whether they are still working in the same way. The above-mentioned manual (in Dutch) has been translated and slightly adjusted by D. Dobler to English. Neither A.W. van der Vaart, nor M.A. Jonker, nor D. Dobler do feel responsible for possible mistakes that result from using this manual.

Contents

1 Manual

This is a short manual for the usage of R in a UNIX-environment. Only the most important possibilities that this programming language is offering is treated here. For more extensive information on R we refer the reader to the official user's manual. Therein it is for example explained how to use character variables, advanced graphics, specialized statistical techniques e.g. analysis of variance (ANOVA), generalized linear models, nonparametric regression, time series analysis, and survival analysis. More extensive information on R can be found on www.r-project.org or with the help of the command

```
> help.start()
```

after you have started an R session.

2 Starting and stopping of R

The R console can be used almost interactively, in the sense that commands can be typed one by one and then one waits for the execution. A *prompt* signalizes that the execution is completed and that R is ready for the next input. In the following we are assuming that R has been started from within X-windows.

Suppose that the UNIX-prompt equals the dollar sign '$', then an R session can be started with the capital letter R.[1]

```
$R.
```

The default R-prompt is the greater than sign '>'. The R-session is terminated with the help of `q()`.

```
> q()
$
```

[1]Note: you could alternatively also use RStudio for commencing an R session.

During the session the execution of code can be interrupted with the help of the usual `CONTROL-C` command.

Each R-command is completed with a `RETURN`. If a command has not been completed after the `RETURN`, R reacts with a *sequel-prompt*: this is by default the plus sign '+'. On the other hand, multiple commands can be typed in just one line by separating them by a semicolon ';'.

```
> 3+3; 1+1
[1] 6
[1] 2
> q( # incomplete command
+ ) # sequel-prompt
$
```

Capital and small letters are interpreted by R in a different way. This is why `c` and `C` are different commands.

3 One exemplary session

The following example session shows some of the possibilities of R. Below you will find only the input and a short explanation of each command. An inspection of the results (after copying the commands into R) gives a quick introduction to R. The used commands will later be explained more extensively.

`$R`	start R session from UNIX-prompt.
`> x <- 1:20`	make a vector `x` with values 1,2,3,. . . ,20.
`> x`	print the value of `x`.
`> y <- rnorm(20)`	generate a random sample of size 20
`>`	from the normal distribution.
`> y <- x + 2*y`	transform `y`.
`> cbind(x,y)`	create a matrix with columns `x` and `y`
	and print the result.
`> x11()`	open a graphics window.
`> plot(x,y)`	plot `y` against `x`.
`> lsfit(x,y)$coef`	compute the coefficients of the least squares line.
`> abline(lsfit(x,y))`	add the least squares line to the plot.
`> lines(spline(x,y))`	also add an exact spline fit.
`> x <- rnorm(50)`	generate a normal sample of size 50.
`> y <- rnorm(50)`	same.
`> plot(x,y)`	plot a scatter diagram.
`> cor(x,y)`	compute the correlation between `x` and `y`.
`> z <- x+y`	compute the sum of `x` and `y` coordinate-wise.
`> cor(x,z)`	compute the correlation.
`> hist(z,prob=T)`	plot a histogram.
`> help(hist)`	give the documentation of the function `hist`.
`> var(z)`	compute the variance of `z`.
`> u <- seq(-5,5,0.1)`	form a row of points between -5 and 5
	with a step size of 0.1.
`> v <- dnorm(u,0,sqrt(2))`	compute the normal density.
`> lines(u,v)`	add a plot of the normal density.
`> {hist(z, xlim=c(-5,5), prob=T}`	repeat the plot of the histogram,
	but don't execute this command yet.
`+ lines(u,v)`	repeat and execute the command.
`> x[x<0]`	select the negative elements in `x`.
`> sum(x<0)`	count the number of negative elements in `x`.
`> u <- seq(-pi,pi,length=100)`	create a uniform lattice of 100 points.
`> plot(cos(u),sin(u),type="l")`	draw a circle.
`> plot(sort(x),1:50/50,`	plot the empirical distribution function of `x`.
`+ type="s",ylim=c(0,1))`	
`> lines(u, pnorm(u))`	add the true distribution function.
`> q()`	quit R.
`$`	the UNIX-prompt.

4 Help

With the help of `help` it is possible to find out right from the terminal what a specific command does. An explanation of the standard function `hist` is obtained as follows

```
> help(hist)
```

A function name that contains special characters must be surrounded by quotation marks.

```
> help("%*%")
```

This calls information on the operation `%*%` (matrix product). For each standard command in R it is possible to find information this way.

5 Vectors

The simplest data structure in R is the vector. A vector is a single object that consists of a row of numbers, text, or logical symbols. A vector named `x` that consists of the four numbers 10, 5, 3, 6 can be made using the function `c`.

```
> x <- c(10,5,3,6)
> x
[1] 10  5  3  6
```

The arrow is the *assigning symbol*; it is typed as a smaller than symbol followed by a dash. (The symbol = plays a different role in R but it could also be used for assigning values.)

The function `c` can take an arbitrary number of vectors and create from them a new vector by concatenation. A single number is interpreted by R as a vector of length 1.

```
> y <- c(x,0.555,x,x) # a vector of length 13 with 11th coordinate 5
```

```
> y
[1] 10.000  5.000  3.000  6.000  0.555  10.000  5.000  3.000  6.000
[10] 10.000  5.000  3.000  6.000
```

As is apparent from the examples given above it is possible to print the value of a vector by typing its name. Here, a bracket with the position of the first following coordinate within the vector is displayed at the beginning of the output. The function `round` gives control over the number of decimals. The second argument is the desired number of digits behind the comma.

```
> round(y,1)   # round after the 1 digit behind the comma
[1] 10.0  5.0  3.0  6.0  0.6  10.0  5.0  3.0  6.0  10.0  5.0  3.0  6.0
```

5.1 Calculations with Vectors: Mathematical Functions

Mathematical operations on vectors are executed coordinate-wise. For instance, `x*x` is a vector with the same length as `x` of which the coordinates are the squared of the corresponding coordinates of `x`.

```
> x
[1] 10  5  3  6
> z <- x*x
> z
[1] 100  25  9  36
```

The symbols for the elementary arithmetic operations are the usual `+,-,*,/`. Powers are obtained through `^` Most of the standard functions in R are available and similarly operate coordinate-wise on the vectors. See Table 1 for the names.

```
>               # continuing from the previous display
> log(x)
[1] 2.302585  1.609438  1.098612  1.791759
> exp(log(x))^2     # equivalent to x^2
[1] 100  25  9  36
```

function name	operation
`sqrt`	square-root
`abs`	absolute value
`sin cos tan`	trigonometric functions
`asin acos atan`	inverse trigonometric functions
`sinh cosh tanh`	hyperbolic functions
`exp log`	exponential function and natural logarithm
`log10`	logarithm with basis 10
`gamma lgamma`	gamma and log-gamma function
`floor ceiling trunc`	round down, round up, and integer part
`round`	round
`sign`	sign

Table 1: Mathematical standard operations

Vectors in one expression do not need to have the same length. If this is not the case, the shortest vector is repeated and concatenated to itself as often as necessary to match the length of the longer vector. A simple example is a binary operation between a vector and a constant.

```
> a <- sqrt(x) + 1
```

In this case the constant 1 (a vector of length 1) is first replaced by a vector with the same length as `x` before the incrementation is done coordinate-wise. In the following example the vector `x` with 4 coordinates is replicated $3\frac{1}{4}$ times in order to reach the same length 13 of the vector `y`.

```
> z <- x*y     # y equals c(x,0.555,x,x) as before
>
Warning messages:
Length of longer object is not a multiple of the length of the shorter
object in:  x * y
> z
[1] 100.00  25.00  9.00  36.00  5.55  50.00  15.00  18.00  60.00  50.00
[11] 15.00  18.00  60.00
```

Because this situation is quite artificial, R gives a warning. Only if the desired effect is achieved with certainty, warnings can be ignored.

The usual order of priority is applied for mathematical operations. For this purpose parentheses "(" and ")" can be used to alter the order of execution. Also in cases of doubt it is recommended to use parentheses. Table 2 give the order of priority for all operators in R.

operation	name	priority
`$`	component selection	HIGH
`[ [[`	coordinate selection	
`^`	power transformation	
`-`	unary minus	
`:`	sequence	
`%name%`	special operation	
`* /`	multiplication and division	
`+ -`	addition and subtraction	
`< > <= >= == !=`	logical comparisons	
`!`	logical negation	
`& | && ||`	and, or	
`<- _ ->`	assignments	LOW

Table 2: Order of priority of R-operations from high to low

5.2 Generation of Lattice Points

Uniform rows of numbers play and important role in the use of countless R-functions. Such *lattices* can be constructed in multiple ways. The fastest operation is the colon :.

```
> index <- 1:20
> index
[1] 1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
```

On the other hand, `20:1` produces the same row, just in a descending order. The function `seq` produces more general sequences. Here one determines either the desired step size or the number of desired lattice points as an argument, in addition to the first and the last lattice point.

```
> u <- seq(-3,3,by=.5)     # a uniform lattice with step size .5
```

```
> u
[1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5 0.0 0.5 1.0 1.5 2.0 2.5 3.0
```

Exactly the same effect is obtained with the following commands.

```
> u <- seq(-3,3,length=13)    # produces 13 lattice points
> u <- (-6):6/2
```

The function `rep` replicates a given vector. If the second argument is a single number, then the first argument is that often repeated.

```
> rep(1:4,4)
[1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
```

The second argument of `rep` may as well be a vector of the same length as the first argument. In that case each coordinate specifies how often the corresponding coordinate of the first vector shall be repeated.

```
> rep(1:4,rep(4,4))
[1] 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4
```

5.3 Boolean and Character-valued Vectors

The coordinates of vectors may not only consist of numerical values but also of the logical symbols `FALSE` ('false') and `TRUE` ('true') or character *strings* (rows of letters and/or symbols). However, each vector is not mixed, in the sense that all coordinate are of the same type. This is why there are numerical, boolean, and character vectors in R.

Boolean vectors are most often created from conditions.

```
> x
10  5  3  6
> y <- x>9       # y[i] is T if x[i]>9
> y
[1] TRUE FALSE FALSE FALSE
```

When one works with boolean vectors, the length of the shortest vector is

adjusted by repetitions to match the length of the longest vector, i.e. the same as for mathematical vectors. The preceding assignment is the same as the unnecessarily long `y <- x>rep(9,4)`.

The logical comparisons are `<, <=, >, >=, ==` for equality, and `!=` for inequality. (The symbol `=` is not a logical operator!) The logical operators *and, or,* and *negation* are given by `&, |, !`. Similarly to mathematical operations, these operations may likewise be applied to vectors.

```
> (3<x) & (x<10)    # do not use (3<x<10)!
[1] FALSE TRUE FALSE TRUE
> (!y)
[1] FALSE TRUE TRUE TRUE
```

The parentheses around `!y` are not always necessary but in this case they may not be left out. An obvious mistake is the abbreviation of the first command to `(3<x<10)`. This expression does not produce the correct result.

Mathematical operations can likewise be applied to boolean vectors. In this case the logical values `FALSE` and `TRUE` are first replaced with 0 and 1. For example, the function `sum(x)` computes the sum of the coordinates of a vector `x`. Hence, `sum(y)` gives the number of `TRUE` in the boolean vector `y`.

```
> sum(x > 9)      # number of coordinates x[i] > 9
[1] 1
```

A *character vector* of length 1 is given in R by a row of characters surrounded with quotation marks such as `"x"` or `"x-values"`. In the following, a character vector of length three is created in two different ways.

```
> x <- c("row a", "row b", "row c")
> x
[1] "row a" "row b" "row c"
> letters[1:3]     # a standard character vector
[1] "a" "b" "c"
> paste("row",letters[1:3])
[1] "row a" "row b" "row c"
```

An alternative way to permanently transform a boolean vector into a numer-

ical vector is to use the function `mode`.

```
> y
[1] TRUE FALSE FALSE FALSE
> mode(y) <- "numeric"
> y
[1] 1 0 0 0
> mode(y) <- "logical"
> y
[1] TRUE FALSE FALSE FALSE
```

5.4 Missing Data

There is just one exception for the rule that vectors must not be of mixed type. The special value `NA` may appear in vectors of any type. This value can be used for missing data. ('`NA`' is an abbreviation for '*not available*'.) In applications of mathematical operations in which the value `NA` appears at least once, the overall result will be `NA` as well. With the help of the function `is.na` one can check which coordinates of a vector have the value `NA`.

```
> x <- c(NA,1,4)
> x
[1] NA 1 4
> x+1
[1] NA 2 5
> is.na(x)
[1] TRUE FALSE FALSE
```

Many functions have a built-in functionality for removing `NA`-values.

```
> sum(c(NA,1,2), na.rm=T)
[1] 3
```

5.5 Subscripts

A part of a vector can be selected according to the generic command

`> x[`*subscript*`]`

The simplest example is the selection of the ith coordinate.

```
> x[i]     # i-th coordinate of x
```

More generally, *subscript* can take one of the following three forms.

- A vector of positive natural numbers. The result is the vector of those coordinates of `x` of which the numbers appear in *subscript*. The coordinates of a vector are always thought of as 1, 2, 3, ...

  ```
  > x
  [1] 10  5  3  6
  > x[1:3] ; x[c(4,1,4)]    # the second gives the vector (x[4],
  x[1], x[4])
  [1] 10  5  3
  [1]  6 10  6
  ```

- A boolean vector of the same length as `x`. The result is a vector with those coordinates of `x` for which the corresponding coordinates of the boolean vector have the value `T`.

  ```
  > y <- x>9     # y is (TRUE, FALSE, FALSE, FALSE)
  > x[y]      # gives x[1]
  [1] 10
  > x[x>9]
  [1] 10
  ```

- A vector of negative numbers. All coordinates of `x` except the specified are selected.

  ```
  > x[-(1:2)]     # gives (x[3], x[4])
  [1] 3 6
  ```

The other way round, assignments to subscripted vectors can be made as well. Here, all unselected coordinates remain unchanged.

```
> x[x<0] <- 0             # sets all negative coordinates to 0
> y[y<0] <- -y[y<0]       # the same result as y <- abs(y)
```

A vector must exist before one selects a part of it. A numerical vector of length `n` can be created using the function `numeric`.

```
> x <- numeric(5)
> x
[1] 0 0 0 0 0
```

Another possibility is to make use of the function `rep`.

6 Functions

All commands in R are executed as *commands*. A bunch of them, such as `c` and `seq`, has already been discussed above. There is a whole lot of standard functions integrated in R. In this manual it is possible to discuss but a small fraction of them. In addition, each R user has to possibility to write his/her own functions. All functions are used in the same way.

A function is called by typing its name, possibly together with some arguments.

```
> functionname(arg1,arg2,...)
```

The parentheses are obligatory, even if the function does not need any arguments. Without the parentheses R reacts by printing the definition of the function.

```
> functionname()  # execute the function functionnname (without arguments)
> functionname    # print the definition of the function functionname
```

The *arguments* can be obligatory or optional. Optional arguments may be left out when the function is called. In such a case, R uses, if necessary, default values. Each argument has a name. Optional arguments are most commonly specified with the help of the expression `argumentname =` *value*. For instance, in

```
> seq(-3,3,length=1000
```

the third argument has the name '`length`'. The advantage of specifying the arguments with their names is that the order of giving the arguments may then be altered and unimportant arguments may be left out. For instance, in the definition of the function `seq`, `by` is the third and `length` is the fourth argument. In a call of `seq(-3,3,1000)` R interprets the number 1000 as the third argument in the definition of `seq`. Since this is `by`, the outcome of the previous command is a different one than that of `seq(-3,3,length=1000)`.

Argument names may be abbreviated to the first few letters as long as they still uniquely specify the argument name.

```
> seq(-3,3,1000)
```

Some functions have a variable number of arguments. One such example is the concatenation function `c`.

6.1 Data-Manipulation

The functions

```
> length(x); sum(x); prod(x); max(x); min(x)
```

respectively give the length of the vector `x`, the sum, product, maximum, and minimum of its coordinates. The latter four functions may be applied to multiple vectors at once.

```
> sum(x,y); prod(x,y) ; max(x,y) ; min(x,y)
```

These commands give the sum, product, maximum, and minimum of all the coordinates of the vectors `x` and `y` put together. For example, the first command is equivalent to `sum(c(x,y))`.

The function `cumsum(x)` produces a vector with the same length as `x` in which the ith coordinate is equal to the sum of the first i coordinates of `x`.

```
> cumsum(rep(1,10))
```

```
[1]  1  2  3  4  5  6  7  8  9 10
```

The numbers in a vector can be sorted with the help of `sort`.

```
> x <- c(2,6,4,5,5,8,8,1,3,0)
> length(x)
[1] 10
> sort(x)
[1] 0 1 2 3 4 5 5 6 8 8
```

Via `order` one obtains the permutation that is necessary for transforming `x` to `sort(x)`. This command can be used to permute a second vector simultaneously with a first one.

```
> order(x)
[1] 10  8  1  9  3  4  5  2  6  7
```

In statistical terms, `sort(x)` is the vector of the *order statistics*; `order(x)` is something different! From the definitions it follows that `x[order(x)]` also gives the order statistics of `x`.

The function `rev` reverses the order of the coordinates of a vector. This is why `rev(sort(x))` is the vector of the order statistics in decreasing order.

```
> rev(sort(x))
[1] 8 8 6 5 5 4 3 2 1 0
> rev(x)
[1] 0 3 1 8 8 5 5 4 6 2
```

The function `rank(x)` gives the ranks of the coordinates of `x`. In the case of ties in `x` mid-ranks are used, as usual in statistics.

```
> rank(x)
[1] 3.0 8.0 5.0 6.5 6.5 9.5 9.5 2.0 4.0 1.0
```

The command `unique(x)` gives the different values in `x` and the boolean vector `duplicated(x)` indicates for each coordinate whether it has already appeared previously.

```
> unique(x)
```

```
[1] 2 6 4 5 8 1 3 0
> duplicated(x)
[1] FALSE FALSE FALSE FALSE TRUE FALSE TRUE FALSE FALSE FALSE
```

Finally, `diff(x)` gives the *spacings* of `x`, i.e. a vector with one coordinate less that specifies the difference of consecutive coordinate pairs of `x`. With the help of optional arguments `diff` might as well produce higher order differences, with a chosen lag.

```
> x
[1] 2 6 4 5 5 8 8 1 3 0
> diff(x)                 # the vector x[2]-x[1], x[3]-x[2], ...
[1] 4 -2 1 0 3 0 -7 2 -3
> diff(x, lag=2)           # the vector x[3]-x[1], x[4]-x[2], ...
[1] 2 -1 1 3 3 -7 -5 -1
> diff(x,lag=1,differences=2)     # the same as diff(diff(x))
[1] -6 3 -1 3 -3 -7 9 -5
```

6.2 Statistical Summaries

A number of functions gives statistical summaries. The names are usually self-explanatory. Some of the functions given in Table 3 are not standard but local functions.

The quantiles function has two vectors as arguments. It gives the empirical quantiles of the first vector for the percentages specified in the second.

```
> quantile(x,c(0.05,0.1))     # the 5% and 10% quantile of x
```

A *stem-and-leaf-plot* is obtained by the function `stem`.

```
> stem(x)
Decimal point is at the |
-2 | 221
-1 | 87
-1 | 443320
-0 | 99888665
```

function name	operation
`mean(x)`	mean
`mean(x, trim=`α`)`	trimmed mean
`median(x)`	median
`var(x)`	variance, covariance matrix
`mad(x)`	median absolute deviation
`range(x)`	the vector `c(min(x),max(x))`
`quantile(x,prob)`	quantiles
`summary(x)`	mean, minimum, maximum, quartiles
`stem(x)`	stem-and-leaf-plot
`cov(x,y)`	covariance
`cor(x,y)`	correlation coefficient
`acf(x,plot=F)`	auto- and cross-covariances and correlations

Table 3: Statistical summaries

```
-0 | 44433221
 0 | 01113334444
 0 | 88899
 1 | 002
 1 | 5669
```

6.3 Probability Distributions and Random Numbers

For each of the most commonly used probability distribution there are four functions available which compute the cumulative distribution function, the probability density/mass function, the quantile function, and random numbers. The names of these functions are always a code for the probability distributions, preceded by the letters:

- `p` (for probability): cumulative distribution function.
- `d` (for density): probability density/mass function.
- `q` (for quantile): quantile function.
- `r` (for random): random numbers.

Code	Verdeling	Parameters	Defaults
`beta`	beta	`shape1, shape2`	-, -
`binom`	binomial	`size,prob`	-, -
`cauchy`	Cauchy	`location, scale`	0, 1
`chisq`	chi-square	`df`	-
`exp`	exponential	`rate`	1
`f`	F	`df1, df2`	-, -
`gamma`	gamma	`shape, rate`	-, 1
`geom`	geometric	`prob`	-
`hyper`	hypergeometric	`m,n,k`	-,-,-
`lnorm`	lognormal	`meanlog, sdlog`	0, 1
`logis`	logistic	`location, scale`	0, 1
`nbinom`	negative binomial	`size,prob`	-,-
`norm`	normal	`mean, sd`	0, 1
`pois`	Poisson	`lambda`	1
`t`	Student's t	`df`	-
`unif`	uniform	`min, max`	0, 1
`weibull`	Weibull	`shape`	-
`wilcox`	Wilcoxon	`m,n`	-, -

Table 4: Codes for probability distributions

For instance for the normal distribution, these are the functions.

```
> pnorm(x,m,s); dnorm(x,m,s); qnorm(u,m,s); rnorm(n,m,s)
```

Here, `m` and `s` are optional arguments which specify the expectation and standard deviation (not variance!); the `n` in `rnorm` is the number of random variables to generate and `x` and `u` are vectors in the domain of the functions. (Thus, the coordinates of `u` are between 0 and 1.) Table4 gives an overview probability distributions for which analogous functions are available, together with the arguments. The digits in the column 'Defaults' are the default values for the parameters.

The function `sample` does random drawings from a given population without replacement. If all elements of the population are drawn without replacement (by default), then the result is a random permutation.

```
> sample(x,3)       # 3 drawings from coordinates of x without replacement
> sample(x)         # random permutation of the coordinates of x
> sample(x,100,replace=T) # 100 drawings with replacement
> sample(c(0,1),100,replace=T,prob=c(0.3,0.7))
        # 100 drawings with replacement, where 0 is drawn
        # with probability 0.3 and 1 with probability 0.7
```

The 'random numbers' are generated according to an algorithm that initiates from the value of the vector `.Random.seed`. The value of this vector is permanently adjusted such that new random numbers are obtained. However, sometimes it is important to reproduce previously obtained results. This is possible by saving the value of `.Random.seed`.

```
> seed <- .Random.seed
> seed
[1] 61 60 42 6 28 3 12 22 32 29 39 2
> rnorm(5)
[1] -1.4643258 1.6484454 -1.1360949 -0.8188030 0.4044431
> .Random.seed <- seed
> rnorm(5)
[1] -1.4643258 1.6484454 -1.1360949 -0.8188030 0.4044431
```

In order to get better ('more random') results it is also possible to let the random number generator start with a number of your own choosing. This could hence have as a result that two simulation studies do not make use of the same row of random numbers. An easy way to make use of this is to type `set.seed` with an arbitrarily chosen natural numbers as the argument.

```
> set.seed(i)
```

7 Graphics

It is one of the most essential purposes of R that its graphical output can be used. Whenever R is started within the X-windows system, a *graphics window* is automatically opened whenever a plot command is used. The output of plot commands appear automatically in that graphics window.

Size and position of the graphics window can be changed in the usual way. After adjusting the size the plot is drawn anew such that the graphic fills the whole window.

There are two sorts of plot commands in R.

- *High level plot functions* produce complete plots and. if necessary, they delete a previous plot output.
- *Low level plot functions* aim to add some graphical output to a previously created plot.

In order to draw two graphics in one plot it is therefore necessary to use one high level and one low level plot function.

The most commonly used plot function is `plot`. Without options this command produces a scatterplot of the coordinates of a first vector against those of a second.

```
> x <- rnorm(50); y <- rnorm(50)
> plot(x,y,main="figuur 1")     # see Figure 1
```

With the option `type="l"` consecutive points are combined by lines. This is what makes `plot` suitable for plotting graphs.

```
> u <- seq(0,4*pi,by=0.05); v <- sin(u)
> plot(u,v,type="l",xlab="",ylab="sin",main="figuur 2") #see Fig.  2
```

Other high level plotting functions create more specific plots. Plots of statistical relevance are the *histogram*, the *boxplot*, and the *qq-plots*.

```
> x <- rnorm(50)
> hist(x, main="figuur 3")     # Figure 3
> text(0,5,"histogram")        # print "histogram" in the point (0,5)
> boxplot(x, main="figuur 4")
> qqnorm(x, main="figuur 5")     # Figure 5
```

The range of the axes in a plot is equal to the coordinates of the vectors `xlim`

and `ylim`. In a simple use it is useful to let R choose the default values for these vectors. However, one could also specify the optional arguments `xlim` and `ylim` in the plot function.

The same holds for the axis labels and main and subtitles. Table 5 gives an overview of some of such optional arguments. Many other aspects of plots can be controlled with the help of `par`. See `help` for more information.

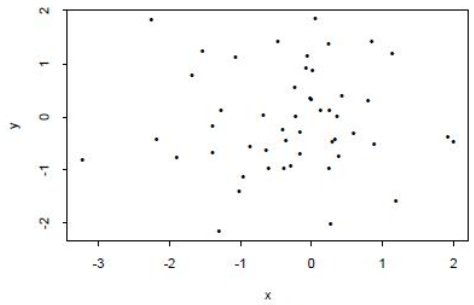
The function `lines` is the low level version of `plot(...,type="l")`. This function can be used for drawing a second graph above an existing one. Consider for instance the plotting of a histogram and the true density of a normal sample in one plot.

```
> x <- rnorm(100)
> range(x)
[1] -3.226397 2.001791 # by default the range in hist(x) becomes (-4,3)
> hist(x,xlim=c(-4,4), xlab="", prob=T, main="Figure 6")
> u <- seq(-4,4,by=0.1)
> lines(u,dnorm(u))
```

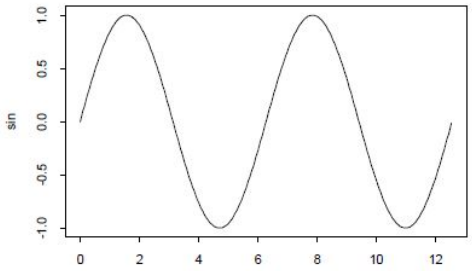
Multiple plots can be plotted beside and above one another in just one plot frame with the help of the function `par`.

```
> par(mfrow=c(r,k))
```

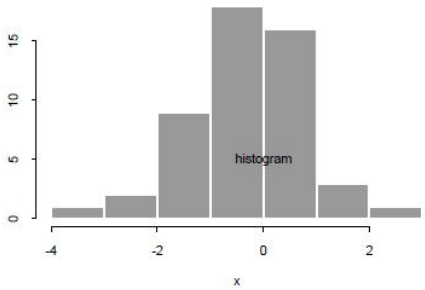
figuur 1



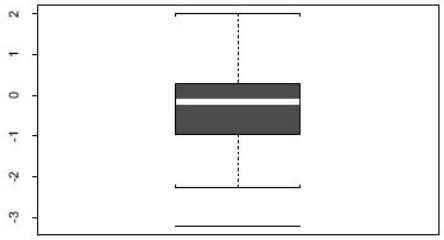
figuur 2



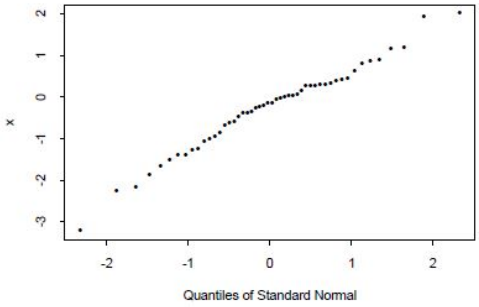
figuur 3



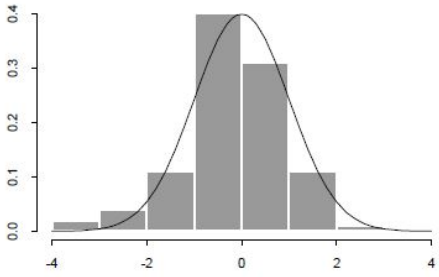
figuur 4



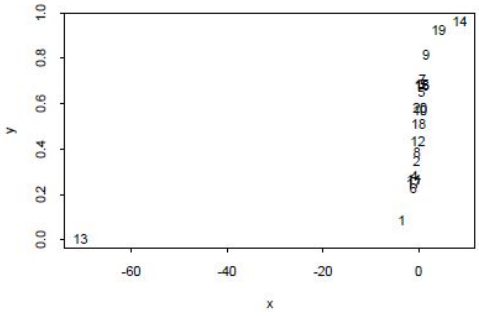
figuur 5



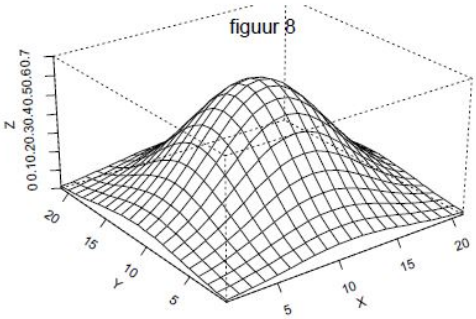
figuur 6



figuur 7



figuur 8



The following first $(r \times k)$ plots are then arranged in one plot frame, in r rows and k columns.

An extra *graphics window* can be opened with the command

```
> x11()
```

The output of the plot commands appear in the new window now.

In order to print a plot or to save it as a postscript file a postscript 'graphics device' has to be opened. The general approach to print plots on the Oce-printer is as follows.

```
> postscript("|lpr -Poce")
> ...
>      plot commands
> ...
> dev.off()     # sends all plots to the printer
```

The plots can be collected and be sent to the printer in one go. The `postscript` command sets the *current graphics device* to postscript. As a consequence, all executed graphical commands that follow are translated into postscript-code and written in a file. The postscript graphics device is terminated via `dev.off`. In that moment the postscript file is automatically sent to the (postscript) printer (only if a default printer command is prepared). By default the paper (a A4 page) is completely filled. This often results in an unattractive relation of the x- and the y-axis. Through the options `height` and `width` this can be improved. For instance,

```
> postscript("|lpr -Poce",he=8,wi=8)      # a plot of 8 x 8 inch
```

The general approach for saving a plot as a postscript-file is

```
> postscript("filename.ps")
>      plot commands
> dev.off()
```

Again, the dimensions of the plots can be adjusted via `height` and `width`.

It is possible to send output alternately to the graphics window and to the

option name	explanation
xlab="*xlabel*"	Label on the x-axis
ylab="*ylabel*"	Label on the y-axis
xlim=c(l,r)	Range of the x-axis from l to r
ylim=c(l,h)	Range of the y-axis from l to h
main="*title*"	Title above the plot
sub="*subtitle*"	Title below the plot
lty=n	Line type: n=1: normal; n=2, 3,...: broken

Table 5: Some plot options

postscript file. It is furthermore possible to open multiple graphics windows simultaneously. One of the graphics devices (a window or postscript) is always the current device and receives the output for the following plot command.

```
> x11()        # opens graphics window
> x11()        # opens second graphics window
> postscript() # opens postscript device
> dev.list()   # gives all open devices
       X11 X11 postscript
         2   3          4
> dev.cur() # the current device
postscript
         4
> dev.set(2) # makes the graphics device 2current
    X11
      2
```

Some High Level Plot functions

`> plot(x,y)`	Scatterplot the points `(x[i],y[i])`.
`> plot(x,y,type="l")`	Plot the points `(x[i],y[i])` with consecutive points connected by a straight line.
`> plot(x,y,type="n")`	Plot the axes, but not the points.
`> plot(x,y,type="s")`	Plot a step function.
`> plot(mat)`	Plot the points `(mat[i,1],mat[i,2])`.
`> matplot(matx,maty)`	Plot the columns of matrices against one another.
`> hist(x)`	Histogram of vector `x`.
`> boxplot(x)`	Boxplot of vector `x`.
`> qqnorm(x)`	qq-plot against the normal distribution.
`> qq`*verd*	qq-plot against the distribution with code *verd*.
`> qqplot(x,y)`	Empirical qq-plot of `x` against `y`.
`> persp(z)`	Quasi three-dimensional plot.

Some Low Level Plot functions

`> lines(x,y)`	Plot the points `(x[i],y[i])` with consecutive points connected by a straight line.
`> title("`*title*`","`*subtitle*`")`	Print titles.
`> text(x,y)`	Plot coordinate numbers at the points `(x[i],y[i])`.
`> text(x,y,label)`	Plot the character string `label[i]` at `(x[i],y[i])`.
`> abline(a,b)`	Add the line `y=a+bx`.
`> points(x,y)`	Add the points `(x[i],y[i])`.
`> polygon(x,y)`	Polygon with nodes `(x[i],y[i])`.
`> mtext("`*text*`",side,line)`	Plot *text* into the margins.
`> legend()`	Add a legend.

Figures 7 and 8 illustrate two other graphical possibilities. They are obtained the following way.

```
> x <- rcauchy(20); y <- runif(20);
> plot(x,y,type="n", main="figuur 7"); text(x,y)
> u <- dnorm(seq(-1,1,0.1),0,0.5)
> z <- outer(u,u)
> persp(z, main="figuur 8")     # a bivariate density
```

Some R-functions make use of the mouse.

With the function `identify(x,y,labels)` one can identify the points in a scatterplot, for instance an outlier.

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

Afterwards, click with the left mouse button at the position of a point. If this is the point `(x[i],y[i])`, then the `i`th coordinate of the character vector `labels` will appear at the position of the point. The default label is `i`. This procedure can be repeated, and stopped with pressing the right mouse button.

The R-command `locator` is useful for obtaining the exact coordinates of a number of points on the screen, for instance, if one wishes to print some text there. Begin by typing:

```
> z <- locator(n)
```

Then click at n points in the graphics window. The coordinates of these points are saved in the list `z`. It is particularly handy to combine `locator` with `text`.

```
> text(locator(1),"Text at the correct position")
```

The text now appears at the position in the graphic where one clicks with the mouse.

After `locator(n=2,type="l")` a line can be printed in the graphic by clicking two points in the graphics window.

8 Lists

A *list* is a roughly speaking a vector of which the components are allowed to be different types of R-objects. One example is a list consisting of a numerical vector, a character vector, another list (sublist) and a function. In the basic usage of R, the list is particularly important because the results of standard functions often consist of a list. Consider for instance the results of the

function `lsfit`. In its simplest use it computes the least-squares regression line of `y` on `x`.

```
> x <- 1:5; y <- x + rnorm(5,0,0.25)
> z <- lsfit(x,y)
> z
$coef:
  Intercept         X
 -0.3286285 1.059901
$residuals:
[1] 1.1562909 -2.1479019 1.4187857 -1.0190293 0.5918546
$intercept:
[1] TRUE
```

In this example the value of `lsfit(x,y)` is assigned to `z`, a list with the first component being a (1×2)-matrix consisting of intercept and slope of the least-squares line; the second component is a vector with the values of the residuals and the third component is a boolean vector of length 1 that indicates if an intercept is included in the model. (Truth be told, the output of `lsfit` has even more components but these are omitted here.) The three components have the names: `coef, residuals,` and `intercept`.

The components of a list can be selected in two ways.

– Via component number: `z[[i]]` is the ith component. Use double brackets!

– Via component name: `z$`*name* is the component with the name *name*.

```
>                # continuing from the previous display
>                # the following 4 commands produce the same output
> z[[2]]
> z$residuals
> z$r
[1] 1.1562909 -2.1479019 1.4187857 -1.0190293 0.5918546
```

In this case the selected component of the list is a vector of length 5. In accordance with the format for selecting coordinates of vectors, one can select a coordinate of that vector.

```
> z$r[4]        # fourth coordinate of z$r = the fourth residual
[1] -1.019029
```

A list can be formed with the help of the command `list`. Here, the names of the components may immediately specified with the help of the =-symbol.

```
> x <- 1:5
> y <- list(numbers=x, trueorfalse=T)
> y
$numbers:
[1] 1 2 3 4 5
$trueorfalse:
[1] TRUE
```

———-

The function `names` retrieves the names of the components of a list. This function could also be used to alter the names.

```
>          # continuing from previous display
> names(y)
[1] "numbers" "trueorfalse"
> names(y) <- c("serial_numbers", "true")
```

Additional components cab be added to a list via assignment to a selected component.

```
>         # continuing from previous display
> y[[3]] <- 1:3; y$four <- "c"
> y
$serial_numbers:
[1] 1 2 3 4 5
$true:
[1] TRUE
[[3]]:
[1] 1 2 3
$four:
[1] "c"
```

9 Matrices and Arrays

A *matrix* can be created from a vector in two ways.

```
>        # the two commands...
> x <- 1:8
> dim(x) <- c(2,4)
>         # ...have the same result as this one command
> x <- matrix(1:8,2,4, byrow=F)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

By using each of the functions `dim` or `matrix` a matrix is by default column-wise filled in with the coordinates of the vector. Within `matrix` this default can be altered by specification of the argument `byrow=T`.

A matrix can also be created by the combination of a number of vectors of the same length, either column- or row-wise, with the help of the functions `cbind` or `rbind`.

```
> cbind(c(1,2),c(3,4))
      [,1] [,2]
[1,]    1    3
[2,]    2    4
```

A matrix may be considered a special sort of vector: a vector with an additional attribute `dim`. All numerical manipulations that apply coordinate-wise to vectors can as well be applied to matrices.

```
> max(x)
[1] 8
> x * x^2
      [,1] [,2] [,3] [,4]
[1,]    1   27  125  343
[2,]    8   64  216  512
```

In addition, there are countless functions that are specifically applicable to matrices. Matrix multiplication is `%*%` and the transpose is `t(x)`.

```
> matrix(1:4,2,2, byrow=T) %*% x
      [,1] [,2] [,3] [,4]
[1,]    5   11   17   23
[2,]   11   25   39   53
>
> t(x)
      [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
```

The function `apply` applies a function, that is given as an argument, to all rows or columns of a matrix.

```
> apply(x,1,mean) # computes the means of the rows of x
> apply(x,2,max)  # computes the means of the columns of x
```

9.1 Matrix-Subscripts

Parts of a matrix `x` can be selected according to the syntax `x[`*`subscript`*`]`. Here, *subscript* may take one of three forms.

– A pair (*rows*, *columns*), where *rows* are the desired row numbers and *columns* are the desired column numbers. 'Empty' collections of rows and/or columns result in selecting all rows and/or columns.

```
> x[i,j]      # the (i,j)th element
> x[i,]       # the i-th row
> x[3,c(1,5)] # the elements in the 3rd row and 1st and 5th column
> x[,-j]      # all except the j-th column
```

– A boolean matrix of the same form as `x`. The result is a vector.

```
> x[x>5]
[1] 6 7 8
> x>5
      [,1]  [,2]  [,3] [,4]
[1,] FALSE FALSE FALSE TRUE
[2,] FALSE FALSE TRUE  TRUE
```

function name	operations
`ncol(x)`	number of columns of a matrix
`nrow(x)`	number of rows of a matrix
`t(x)`	transpose of a matrix
`diag(x)`	make a diagonal matrix from a vector or form the diagonal vector of a matrix
`col(x)`	matrix of column numbers
`row(x)`	matrix van row numbers
`cbind(...)`	combine columns to a matrix
`rbind(...)`	combineer rows to a matrix
`apply(x,1,func)`	apply the function `func` to the rows of the matrix
`apply(x,2,func)`	apply the function `func` to the rows of the matrix
`%*%`	matrix multiplication
`solve(x)`	inverse of a matrix
`solve(A,b)`	solution to the system of equations $Ax = b$
`eigen(x)`	eigenvalues
`chol(x)`	choleski decomposition
`qr(x)`	q-r decomposition
`svd(x)`	singular value decomposition
`cancor(x,y)`	canonical correlation
`var(x)`	covariance matrix of the columns

Table 6: Some matrix functions

Arrays

Matrices are two-dimensional arrays. The elementary operations on arrays are analogous to those on matrices.

———-

The function `dimnames` contains the rows and columns of a matrix.

```
> a                     # a character matrix
          sex height weight
Jansen    "M" "1.79"   "75"
Pietersen "V" "1.61"   "58"
Klaassen  "V" "0.80"   "23"
> dimnames(a)             # a list with two character vectors
[[1]]:                    # as components
[1] "Jansen" "Pietersen" "Klaassen"
[[2]]:
[1] "sex" "height" "weight"
> dim(a)
[1] 3 3
```

10 Data-Frames

The elements of a matrix as discussed in the previous section all must have the same `mode`: for instance, numeric, character, or logical. A `data.frame` is essentially a matrix of which the rows have different `modes`. This data structure is particularly interesting in combination with the `read.table` command in order to read tables. (See below.) A `data.frame` can also be made with the help of the command with the same name.

```
> x <- c("M","V","V")
> y <- c(1.79,1.61,0.80)
> z <- c(T,F,T)
> data.frame(x,y,z)
  x    y     z
1 M 1.79  TRUE
2 V 1.61 FALSE
3 V 0.80  TRUE
```

11 Input and Output

For the input of big data sets it is not recommended to use the function `c`. The best way to do this is to first type in the data in a `UNIX`-file with the help of an arbitrary text editor and then to use in R the function `scan` or the function `read.table`. The advantage is that typos can be easily corrected with the help of the text editor.

11.1 How to Read Vectors

The easiest way for inserting daa is via the function `scan`.

```
> x <- scan("file_name")
```

Here, *file_name* is the corresponding`UNIX`-text file. The quotation marks are obligatory. In the command above the function `scan` reads the numbers row by row, where the numbers must be separated by either space character(s), tab, or a new line. Other separators can be specified with the help of an option. The 'text' file is only allowed to contain numbers this basic form of the function. Character strings can be read with the help of another option. The result is always a vector, but of course it is possible to immediately convert it into a matrix.

```
> x <- matrix(scan("file_name"),10,3)  # a file with 30 numbers
```

`scan` can sometimes also be useful for the input of small datasets. If no file is passed as an argument to the `scan` function, an input from the keyboard is expected.

```
> x <- scan()
1:  1 2.5 3.14159  (typed in via keyboard)
4:                                (RETURN)
3 items read
> x
[1] 1.00000 2.50000 3.14159
```

11.2 How to Read Tables

In many cases a data set consists of a table in which the numbers or text in each row refer to a single object. For instance, a UNIX-file could have the following structure.

```
          sex height weight
Jansen    M   1.79   75
Pietersen F   1.61   58
Klaassen  F   0.80   23
```

The different entries of the tables are here placed neatly below one another, but it is only important that they are separated by white space. The command `read.table` converts such a table into the data structure `data.frame`.

```
> z <- read.table("file_name")
> z
          sex height weight
   Jansen   M   1.79     75
Pietersen   F   1.61     58
 Klaassen   F   0.80     23
> dim(z)
[1] 3 3         # z is a 3x3 matrix!
> dimnames(z)
[[1]]:
[1] "Jansen" "Pietersen" "Klaassen"
[[2]]:
[1] "sex" "height" "weight"
```

In a file with a given structure, `read.table` interprets the first row and column as the names of the names of the rows and columns, respectively. This explains why in the resulting `data.frame` in the example given above has the dimensions 3×3 and not 4×4. This can be prevented with the help of an optional argument.

In most cases R is used interactively, but sometimes it is handy to execute a smaller or larger series of commands at once. This can be achieved with the help of `source`. First, create a UNIX text file which contains the desired

commands, typed in the same way as if they had been typed during an interactive R-session (without the R-prompts). Save this file under an arbitrary name, for instance *file_name*, then give the R-command.

```
> source("file_name")
```

The commands standing in the text file are now executed by R one by one and the R-prompt will appear again after the last command has been completed. Below there is an example of a source file.

```
m <- numeric(1000)
for (i in 1:1000) {x <- rnorm(50);  m[i] <- mean(x)}
s <- var(m)
```

11.3 Data Editing

An error in an existing vector, matrix, or data.frame `x` can be mended by exporting the data set via the `write` command as a `UNIX`-file (in ASCII-format), altering the data with an editor (such as `vi` or `emacs`), and finally re-reading the file with the help of one of the methods given above.

```
> x                       # a 3x4 matrix
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> write(t(x),"file_name_2",ncol=4)
```

12 How to Write your own Functions

Writing functions in R is easy. The definition can usually be made quite short if already implemented R-functions and data structures are used. The geenral definition of a function is:

```
functionname <- function(arg1, arg2, ...)    definition
```

Here, `functionname` is a name of your choosing, `function` a fixed keyword for indicating that a function definition is taking place, `arg1, arg2, ...` are argument names of your choosing, and *definition* is a group of R-statements. The value of the last expression in this group is going to be the (output) value of the function. This may be a vector, a list, or any other R-data structure. The R-statements are either one single command or a series of commands that are surrounded by curly brackets. For example, the function `sdev`

```
> sdev <- function(x) sqrt(var(x))
```

computes the sample standard deviation. The *definition* part here consists only of a single statement and calls two other functions. A more involved example is the sample kurtosis.

```
> kurtosis <- function(x){
+  fourthmoment <- sum((x - mean(x))^4)/(length(x)-1)
+  secondmoment <- var(x)
+  fourthmoment/(secondmoment^2)
}
```

It is possible to alter the definition in several ways. First, it is possible to type in a new definition from scratch. However, it is easier to make use of a text editor. After

```
> kurtosis <- vi(kurtosis)
```

the `UNIX`-editor `vi` temporarily takes over from R and the function definition can be altered from within `vi`. Save the eventual text in `vi` and exit `vi`. The continued R session the new definition is used henceforth.

Another way to make use of an editor is to first dump the function in a `UNIX`-file, edit the `UNIX`-file with the help of an editor, and then source the file from within R. This is particularly handy within a window system because R and the editor can then be used in different windows. The general approach is as follows.

```
> dump("function_name", "file_name")
...
(edit the file file_name with your favorite editor, outside of R)
```

```
...
> source("file_name")
```

Following the same idea it is also possible to insert a new function into R. First create (outside of R) a text file with the help of an editor that includes the definition of a function. Then use `source` in order to create the function in R. The 'dumping' is now left out, and `dump` is also not necessary for alterations as long as the original text file exists.

Each argument in a function call is either obligatory or optional. This difference is made in the execution as well as in the definition. First of all, an argument that is not used in a particular execution of the function does not need to be given. Furthermore, an argument is permanently optional if a default value is given in the definition of the function. This happens with the help of the construction `arg =` *default_value* in the definition.

```
> power <- function(x, k=2) x^k   # the default value for k is 2
>
> power(5)    # the default value is used
[1] 25
> power(5,3)  # the default value is neglected
[1] 125
```

Assignments within a function defintion are local for the '*frame*' of the function. This means that the direct results of such assignments are lost after the complete execution of the function. Only the values of the last lone are transferred to the main program. Thus, auxiliary variables within the function definition may have the same names as variables in the main program. The other way round, all relevant information of the function should be placed in the last line.

```
> x <- 0
> lost <- function(x) x <- 3; x
> lost(x)
[1] 3
> x
[1] 0
```

In the awkward case that a global and permanent assingment shall be made

from within the definition of a function, one can use the `<<-` operator.

```
> kept <- function(x) x<<-3
> kept(x)
[1] 3
> x
[1] 3
```

The arguments of a function may be of any R-object type; thus also functions. For instance, if `qdistribution` specifies the quantile function of a generic probability distribution, then the following function gives a generator of random numbers drawn from the distribution.

```
> rdistribution <- function(n,qdistribution){
+  u <- runif(n)
+  qdistribution(u)}
```

This definition makes sense even if there is no function with the name of `qdistribution` since `qdistribution` is just the formal name of the second argument of `rdistribution`. In the call of `rdistribution`, however, it is mandatory to fill in a real function for `qdistribution`.

```
> rdistribution(5, qnorm)
[1] 0.9332848 2.2774360 -0.3208976 0.1994874 -0.5723007
> rverdeling(1000, qeinstein)
Error:  Object "qeinstein" not found
Dumped
```

(By the way, `rnorm` is a standard R-function.) If other functions are called in the definition of a function, then the arguments can be passed along. A more general form of the random number generator is as follows.

```
> rdistribution <- function(n,qdistribution,mean=0,sdev=1){
+  u <- runif(n)
+  qdistribution(u, mean, sdev)}
```

However, this function is only appropriate in combination with quantile functions that require exactly two arguments. A better possibility is to specify the arguments implicitly.

```
> rdistribution <- function(n,qdistribution, ...){
+  u <- runif(n)
+  qdistribution(u, ...)}
```

The meaning of the (precisely) three points is that each argument that is given in the actual call of `rdistribution` is in the same order forwarded to `qdistribution`. Given the last definition of `rdistribution`, the following two calls are now acceptable.

```
> rdistribution(5, qnorm,10,1)  # qnorm expects 2 arguments
> rdistribution(5,qpois,2)      # qpois expects 1 argument
```

13 If, For, While, Repeat

The R-language has a primary construction and possibilities for loops that are analogous to those of other programming languages. The `if` function in R has two forms.

```
> if (condition) expr1
> if (condition) expr1 else expr2
```

In both cases the *condition* is evaluated first; if the value is `TRUE`, then *expr1* is computed. If the value of the condition if `FALSE`, then nothing happens in the first `if` statement; in the second statement *expr2* is executed then. Do not forget to place the parentheses around *condition*!

If the *condition* results in a boolean vector of length more than one, then R gives a warning but nevertheless executes the `if` statement with the first coordinate of the boolean vector.

In the definition of a function the construction `if (!missing(`*argument*`))` can be used in order to test whether a particular argument has been specified in the call of the function, or not.

From the previous sections it is apparent that R executes many loop-iterations implicitly. For example, the command `sqrt(x)` automatically applies the square-root function to each of the coordinates of the vector `x`. By mak-

ing efficient use of this it is often possible to avoid explicit uses of loop-constructions. Not only can the R code be shortened by this, but the code is usually also executed much faster in many cases. The explicit loop-constructions that are now going to be discussed are relatively slow in R.

The most important loop-construction is the `for`-construction.

`> for (name in` *`values`*`)` *`expr`*

The result is the one-by-one assignment of the elements of *values* to `name` and for each time the evaluation of *expr*. Here, *values* is usually a vector or a list. The quantity `name` is a dummy that will not exist anymore after the `for` statement.

```
> x <- y <- NULL     # NULL is an empty object
> for (i in 1:10) {x <- c(x,i); y <-c(y,0)}
> x
[1] 1 2 3 4 5 6 7 8 9 10
> y
[1] 0 0 0 0 0 0 0 0 0 0
```

An alternative is the `while`-statement.

`> while (condition)` *`expr`*

The execution starts with the testing of the *condition*. If the value is *FALSE*, then the execution stops; if the value is `TRUE`, then *expr* is evaluated and a new iteration starts from the beginning.

```
> x <- numeric(10)
> x
[1] 0 0 0 0 0 0 0 0 0 0
> i <- 0
> while (i < 10) { i <- i+1
+  x[i]<- i}
> x
[1] 1 2 3 4 5 6 7 8 9 10
```

It should be mentioned that the previous examples did not realize a good

programming style in R. Compare this with the command.

```
> x <- 1:10
```

The `repeat`-construction can be used in combination with `break`. The general form is

```
> repeat {
...
if (condition) break
...
}
```

When R evaluated `break`, then the innermost `for, while`, or `repeat` loop that contains `break` is quit. Besides `break` it is also possible to use `next`. This has as a result that the following iteration of the loop is immediately commenced, without the completion of the present one.

14 Data Directories

During the termination of R one is asked whether the workspace should be saved. If y (for yes) is typed, then all objects, that were created during the R-session, are saved in the file '.RData' in the directory where R has been started. Thus, they are present again during the next R-session if R is started from the same directory once again. Technically speaking, for each R-object there is one corresponding `UNIX`-file with the same name as the R-object. This means that the file can be deleted, copied, have its name changed, etc, via the appropriate `UNIX`-commands, However, the files in the `.Data`-directory are not text files and they cannot be read from outside of R: in order to accelerate the processing in R all R-objects are coded in a special format. One can make readable prints of R-objects with the help of R-commands `write` and `cat` (for data-objects) and `dump` (primarily useful for functions).

In R one gets via the `ls` command and overview of all objects in the current `.Data`-directory. Unnecessary objects can be removed via `rm`.

```
> ls()
> ls("x*")      # prints all objects whose names begin with x
> rm(x,y,object)  # removes x, y, and object
```

Both operations can outside of R also be executed with the help of `UNIX`-commands of the same name. Wild cards are allowed.

Besides the objects that the user has created, there is a large number of standard R-objects, mainly functions, in directories somewhere in the system. R continuously searches through relevant directories in order to find the requested objects. The search takes place in a fixed order, the *search list*. By default it consists of the `.Data`-directory of your home directory.

R stops searching as soon as the named object has been found. One consequence is that a standard function with a particular name (in the R-functions directory) will be unusable from that moment when there is a function with the same name in the `.Data`-directory. The remedy for the dubble use of a name is to delete the self-made object. You don't have to worry that the standard function might be deleted: R would always deny the execution of such commands.

15 Categorical Data

A *category* is a data structure in R that can be used for the analysis of categorical data: data of which the value space is a collection of labels. The function `cut(x,breaks)` creates a category from a numerical vector `x` by checking for each of the coordinates of `x` in which of the intervals defined by `breaks` they are contained.

```
> x
[1] 3 3 5 1 5 5
> breaks
[1] 0 2 4 6 8
>         # the intervals are (0,2], (2,4], etc.
>         # x[1] is in the second interval, x[3] in the third, etc
> cut(x,breaks)
```

```
[1] (2,4] (2,4] (4,6] (0,2] (4,6] (4,6]
Levels:  (0,2] (2,4] (4,6] (6,8]
```

The function `cut` can be combined with `table` for getting the numbers of coordinates of `x` that are in the different intervals.

```
> table(cut(x,breaks))
(0,2] (2,4] (4,6] (6,8]
    1     2     3     0
```

A category can be considered a vector with an addiotional attribute: *levels*. This means that one can make the same computations with a category as with vectors.

```
> table(cut(x,breaks))+10
(0,2] (2,4] (4,6] (6,8]
   11    12    13    10
```

The function `table` can also be used in order to create higher dimensional tables. Given two vectors of categories `x` and `y` with the same length, `table(x,y)` produces a $(k \times r)$ -matrix with as the (i, j)th element the number of pairs `(x[l],y[l])` that have level i in `x` and level j in `y`. Here, the levels of a vector are the same as the ordered different values.

```
> x
[1] 3 3 5 1 5 5
> y
[1] 0 1 0 0 1 1
> table(x,y)
  0 1
1 1 0
3 1 1
5 1 2
> table(cut(x,breaks),y)
     y
      0 1
(0,2] 1 0
(2,4] 1 1
(4,6] 1 2
```

```
(6,8] 0 0
```

The function `tapply(x,category,func)` can be used to apply a given function `func` to all coordinates of a vector `x` of the same level, for each level. The levels are given in the second argument `category`: `x[i]` is thought to have level `category[i]`. The operation is best understood in two steps. First, all coordinates of `x` are sorted per level. Next, the function is applied to each group of coordinates with equal level (conceived as a subvector of `x`). The result is a vector the length of which equals the total number of different levels.

```
> x
[1] 3 3 5 1 5 5
> breaks
[1] 0 2 4 6 8
> tapply(x,cut(x,breaks),sum)
(0,2] (2,4] (4,6] (6,8]
    1     6    15    NA
> y
[1] 0 1 0 0 1 1
> tapply(x,y,sum)
0 1
9 13
```