Chapter 2
Brief Introduction to the R language

2.1 Introduction

Whilst most programs nowadays use interactive graphical user interfaces, R is mainly command-driven. The user has to type in sequences of commands and to request R to execute them. This is something one needs to get accustomed to, however, it has the advantage that the user can be more flexible for creative and complex tasks such as the development of simulation models. With menu-based programs we much depend on the particular organisation of that program (Braun and Murdoch, 2007) and has limited choices.

The following pages have been designed to ease the process of getting used to the command-line approach of the R software. They are a hands-on introduction for readers who are unfamiliar with R to offer just enough material to get started and to motivate beginners to explore and experiment with the program.

Download R from `http://cran.r-project.org` and select the "CRAN mirror" for your country of residence. Click on the download link for your operating system and on the subfolder "base". It is sufficient for the purpose of this book to download and install a binary (precompiled) version. Then click on the download link and launch the installation program. (Double) clicking on the R icon will launch the program.

With no prior knowledge of R it is recommended to follow the examples below during the course of 1-3 days to get started. The introductory chapters of Braun and Murdoch (2007), Bolker (2008), Dalgaard (2008), Jones *et al.* (2009) and Robinson and Hamann (2010) will help to expand the basic knowledge given in this appendix.

2.1.1 Basics

The greater-than sign (`>`) is the prompt symbol indicating that R is ready for user inputs. When this appears in the console window you can type commands. For example, you can simply use R as a calculator in the following way

```
> 1 + 2
[1] 3
```

The number in square brackets in the output line gives the index number of the first element. Operators can also be nested, e.g.

```
> (8 - 5) * 3
[1] 9
```

first 5 is subtracted from 8 and then multiplied by 3.

Although all possible R commands can be typed and run at the prompt, it is recommended to write scripts in the document window of the graphical user interface (GUI). They can be saved on your computer drive (commonly with the extension `*.R`) and will eventually serve as a valuable collection of numerical recipes that you may like to come back to at a later stage. Scripts can be exchanged between colleagues thus enhancing professional collaboration. Also saved R scripts are a good reproducible documentation and repository of your data manipulation and analysis work. They can for example be attached to BSc, MSc or PhD theses and scientific papers as supplementary material.

In the R console, commands are executed by hitting the `Enter` button. By contrast, commands typed in the R script editor need to be highlighted followed by pressing `Ctrl-R` in MS Windows and `Cmd-Enter` in Mac OS for execution. Alternatively you can click on the corresponding icon in the GUI. It is possible to run every code line separately and also to highlight several lines and to execute them at the same time.

Continuing lines are automatically indicated by the + sign when the first line is incomplete.

```
> 5 *
+ 2^3
[1] 40
```

Often the appearance of + in the console means that parantheses or quotes need to be balanced (Robinson and Hamann, 2010, p. 7). A summary of basic operators and functions can be found in Table 2.1.

Any alphanumeric character including "_" and "`.`" (but no spaces) that start with a letter can freely be selected as variable names (Bolker, 2008, p. 23), however, some letters and words have been reserved by R and should be avoided, e.g. "`F`" (= `False`), "`T`" (= `True`) and "`data`". Values are assigned to variables using the `<-` sign, the assignment operator.

As demonstrated in the second and third line of the code below, *x* has the value of 3 and can be used in subsequent calculations.

Table 2.1 Basic arithmetic operators and functions.

Operator/function	Description
`+, -`	Addition and subtraction
`*, /`	Multiplication, division
`^`	Power
`abs()`	Absolute
`exp()`	Exponential function
`log()`	Natural logarithm
`logb(x, base = 2)`	Logarithm to the base 2
`sqrt()`	Square root
`round()`	Rounds decimal numbers
`floor()`	Rounds decimal numbers down
`ceiling()`	Rounds decimal numbers up
`sin(), cos(), tan()`	Trigonometric functions
`integrate()`	Calculates the integral of a function
`<, >`	Smaller, greater than
`==, !=`	Equal, unequal
`<=, >=`	Smaller equal, greater equal
`&, |`	And, or
`is.na()`	Missing?
`round(x, digits = 3)`	Rounding to three decimal digits

```
> x <- 3
> x
[1] 3
```

Typing and running the variable name again will allow you to see its value. Alternatively you can execute the command `print(x)`. Inside loops (see Section 2.1.5) only the command `print(x)` produces the desired output in the R console.

Comments are useful to remember what you had in mind when writing your code. They should be marked with `#` (hash key) to inform the compiler that what follows in the corresponding line is non-executable code that should be ignored. The hash key can also be used to disenable lines of code.

```
> sin(pi / 2) # This is a comment.
[1] 1
```

Also it is important to be aware of the fact that R is case-sensitive, i.e. variable names involving lower-case and upper-case versions of the same letter essentially constitute different variables and confusing them can cause errors:

```
> dbh <- 16.5
> Dbh
Error: object 'Dbh' not found
```

Almost all objects in R are internally represented by vectors even if they contain only one number. The round brackets in the listing below force R to print the vector `dbh` after assigning the value of 5.6 to it. This saves typing another command line for displaying the value of `dbh`.

```
> (dbh <- 5.6)
[1] 5.6
```

In R, functions always require round brackets containing no, one or more arguments and return results without any additional prompt. Using the function `c()` (= concatenate) you can easily create vectors with multiple values, e.g. a *dbh* list. A strength of R is that it can handle entire data vectors as single objects (Dalgaard, 2008, p. 4).

```
> (dbh <- c(5.6, 10.3, 14.5, 27.8, 48.5))
[1] 5.6 10.3 14.5 27.8 48.5
```

Vectorisation and vector arithmetics are an important feature of R to make data processing more efficient. `length(dbh)` returns the length of a vector. The command `mode(dbh)` gives you the data type.

```
> length(dbh)
[1] 5
```

To access a particular stem diameter of the vector `dbh`, e.g. the second one, type

```
> dbh[2]
[1] 10.3
```

Using the function `c` again allows selecting several values at the same time.

```
> dbh[c(2, 4, 5)]
[1] 10.3 27.8 48.5
```

In R, square brackets are always applied to select a subset of data. Negative indices can be used to suppress certain elements. You can for example select all but the second diameter:

```
> dbh[-2]
[1] 5.6 14.5 27.8 48.5
```

It is also possible to print all stem diameters larger than 15 cm with the following code.

```
> dbh[dbh > 15]
[1] 27.8 48.5
```

The same result you obtain by using the command `subset(dbh, dbh > 15)`. Vectors can be manipulated by using basic arithmetic operators, e.g. we can let the stem diameters grow using a growth multiplier of 1.3 (see Chap. 1.1).

```
> dbh * 1.3
[1] 7.28 13.39 18.85 36.14 63.05
```

As a result all elements of the vector `length(dbh)` have been multiplied by 1.3. Naturally vectors can be modified by any other operator or mathematical function. By the way, data can also be read from screen by using the following command:

```
> dbh <- scan()
1:
```

Numbers followed by colons then prompt you to input data in the console.

We can now apply the first summary characteristics to vector `dbh`. For this purpose we use functions that are written in a similar way as in MS Excel and other programs. Minimum and maximum *dbh* we can for example calculate in the following way:

```
> min(dbh)
[1] 5.6
> max(dbh)
[1] 48.5
> range(dbh)
[1] 5.6 48.5
```

A summary including similar characteristics can be achieved with the command `summary(dbh)`.

```
> summary(dbh)
  Min.  1st Qu. Median Mean 3rd Qu. Max.
  5.60  10.30 14.50 21.34 27.80 48.50
```

The `range` command produces minimum and maximum at the same time. `min`, `max` and `range` are functions that are part of the common functionality of R. More sophisticated descriptive statistics include the arithmetic mean, the median, the standard deviation and the coefficient of variation as given in the code below.

```
> mean(dbh)
[1] 21.34
> median(dbh)
[1] 14.5
> sd(dbh)
[1] 17.29026
> sd(dbh) / mean(dbh)
[1] 0.8102276
```

However, when using concrete numbers as opposed to variables you need to resort to the concatenate function, e.g. `mean(c(4, 6, 8, 9))` to obtain `6.75`. Other frequently used statistical functions are provided in Table 2.2.

Table 2.2 Basic arithmetic operators and functions.

Operator/function	Description
`var(x)`	Variance of *x*
`quantile(x, p)`	*p* quantile of *x*
`cor(x, y)`	Correlation between *x* and *y*
`prod(x)`	Product of *x*
`sum(x)`	Sum of *x*
`diff(x)`	Vector with differences `x[i] - x[i - 1]`

Another frequent application in statistics is the ranking of observations according to their size. Consider a vector with various stem-diameter values:

```
> dbh <- c(27.8, 5.6, 48.5, 10.3, 14.5)
> rank(dbh)
[1] 4 1 5 2 3
```

Function `rank()` provides the required ranking. Often we can have observations with the same values, so-called "ties". Using the default of the `rank()` function we obtain

```
> dbh <- c(27.8, 5.6, 5.6, 10.3, 14.5)
> rank(dbh)
[1] 5.0 1.5 1.5 3.0 4.0
```

average ranks for stem diameters 2 and 3 (both 5.5 cm) whilst method `"min"` gives us

```
> dbh <- c(27.8, 5.6, 5.6, 10.3, 14.5)
> rank(dbh, ties.method = "min")
[1] 5 1 1 3 4
```

equal ranks for both observations.

In analogy to numeric vectors character vectors are also possible. However, the text strings need to be wrapped in quote symbols. Consider the following code:

```
> myNames <- "Peter"
> myNames <- c(myNames, "John", "Lucy")
> myNames
[1] "Peter" "John" "Lucy"
```

In the first line, Peter's name is stored to vector `myNames`. Then the idea has come up to add John's and Lucy's name to the vector `myNames`, however, without overwriting the current contents which is Peter's name. That is why the vector `myNames` itself is included on the right hand side of the assignment.

Finally it can be useful to create simple vectors by using the `seq` and `rep` functions. The command

```
> seq(1, 17, by = 2)
[1] 1 3 5 7 9 11 13 15 17
```

creates a sequence of odd numbers between 1 and 17. The optional parameter `by` specifies the step width.

We can now assign the sequence of odd numbers to a variable, `ad`, and use the function `rev` to reverse the order of elements (lines 1-2). The same effect can be achieved by reversing the index using the command `ad[length(ad) : 1]` (line 4):

```
1 > ad <- seq(1, 17, by = 2)
2 > rev(ad)
3 [1] 17 15 13 11 9 7 5 3 1
4 > ad[length(ad) : 1]
5 [1] 17 15 13 11 9 7 5 3 1
```

The command `rep(7, 7)` on the other hand repeats the value '7' 7 times:

```
> rep(7, 7)
[1] 7 7 7 7 7 7 7
```

Simple sequences can also be produced by the following command:

```
> c(1, 7 : 9)
[1] 1 7 8 9
```

The sample function `sample` can be used to return a random permutation of a vector. Consider a vector `x` of integers from 1 to 10.

```
> x <- 1 : 10
>
> sample(x)
 [1]  4  3  6  5 10  2  9  7  1  8
```

Incidentally the command `sample(10)` has exactly the same effect. Using the `size` argument we can now draw a subsample of vector `x`.

```
> sample(x, size = 4)
[1] 10  9  1  7
```

Finally, the `replace` argument allows us to perform sampling with replacement, i.e. each number of vector `x` can be drawn more than once. In that case the sample size can also exceed the length of the vector.

```
> sample(x, size = 20, replace = T)
 [1]  2  2  4  2  8  3  4  4  4  8 10 10  5  1  3  3  2  6  4  1
```

Help and explanations relating to any function or operator, e.g. `log` can be called on by typing `help(log)` or `?log`. For code examples only you can type `example(log)`. There is also a growing community of R users throughout the world and carefully passing your R related question on to an internet search machine may give you quick and useful answers.

`Ctrl-C` or `Esc` stops any processing, which currently is underway. Incidentally, earlier commands can be retrieved in the console by pressing the arrow keys (up and down) of the keyboard. The reasons for any error message can be analysed with the function `traceback()` and to quit your R session type

```
> q()
```

After running `q()` R enquires whether it should save the workspace image, i.e. the data and the sequence of commands of the current session. In the author's experience this is usually not necessary as every session can easily and quickly be reconstructed from the script used so that it suffices to save the script file.

2.1.2 Data frames

Consider two tree variables of interest, stem diameter, *dbh* and total tree height, *h*. The corresponding data columns can be represented by and stored in two separate columns or vectors, `dbh` and `h`.

```
> dbh <- c(33.5, 40.5, 39.0, 54.0, 38.8, 32.4, 29.7, 31.0,
+ 55.5, 26.1)
> h <- c(28.9, 30.0, 26.6, 30.7, 27.0, 26.2, 27.8, 25.5, 31.7,
+ 24.8)
```

```
5  > myData <- data.frame(dbh, h)
6  > str(myData)
7  'data.frame': 10 obs. of 2 variables:
8   $ dbh: num 33.5 40.5 39 54 38.8 32.4 29.7 31 55.5 26.1
9   $ h : num 28.9 30 26.6 30.7 27 26.2 27.8 25.5 31.7 24.8
```

In lines 1-4, 10 stem diameter and corresponding height values are assigned to the vectors `dbh` and `h`. Note that for successful calculations both vectors should have the same length. Also the order of *dbh* and *h* values matters so that corresponding values should occur at the same locations in the two vectors. As they form a common data set, the vectors can be combined as two columns in a data frame entitled `myData` (line 5). The command `str(myData)` is useful to check up on or to remind ourselves of the structure of the data frame. For accessing individual columns you now have to use the name of the data frame plus the column name separated by a `$` sign, e.g. `myData$h`.

A data frame is a two-dimensional structure for storing vectors. Each column corresponds to a variable and each row to an observation.

A common task in working with data sets is to sort the data in an ascending or descending way. This can be accomplished using the following code, in which we sort descendingly according to stem diameter.

```
 > (myData.s <- myData[order(myData$dbh, decreasing = TRUE), ])
     dbh    h
9   55.5 31.7
4   54.0 30.7
2   40.5 30.0
3   39.0 26.6
5   38.8 27.0
1   33.5 28.9
6   32.4 26.2
8   31.0 25.5
7   29.7 27.8
10  26.1 24.8
```

The comma in the command ensures that all columns are included in the sorting process. Again the round brackets prompt R to print the data in `myData.s` on screen.

For checking up on your computations you can list a subset of the total data frame by using indices, e.g. the code

```
 > myData.s[1 : 5, ]
   dbh    h
9 55.5 31.7
4 54.0 30.7
2 40.5 30.0
3 39.0 26.6
5 38.8 27.0
```

gives you the first five rows of the data set `myData`. Round brackets in this case are not necessary as the subsetting induced by the commands in the square brackets is similar to the application of a function. The comma at the end of the expression in the squared brackets indicates that you wish to display all columns of the data set.

Alternatively use the commands `myData.s[2]` or `myData.s["h"]` to select the tree height column only.

`sapply()` is one of many functions in R that have been designed to work with vectors by using *implicit loops* (Dalgaard, 2008, p. 26), which – if programmed in R – would otherwise consume a lot of computation time. `sapply()` allows us to apply a function of our choice (including user-defined ones) simultaneously to all columns of a data frame. In the listing below we calculate the arithmetic mean of both data columns.

```
 > sapply(myData, mean)
   dbh h
 38.05 27.92
```

Alternatively the commands `apply(myData, 1, mean)` and `apply(myData, 2, mean)` calculate the row and column means, respectively.

A new column or vector within a data frame can simply be created by providing a new vector name and assigning values to it. We can, for example, calculate the h/d ratio (see Chap. 1.1) in the following way:

```
 > myData$hd <- 100 * myData$h / myData$dbh
 > myData[1 : 3, ]
    dbh    h       hd
1 33.5 28.9 86.26866
2 40.5 30.0 74.07407
3 39.0 26.6 68.20513
```

A selection of data can be achieved by using a relational expression, e.g.

```
 > myData[myData$dbh > 35, ]
    dbh    h       hd
2 40.5 30.0 74.07407
3 39.0 26.6 68.20513
4 54.0 30.7 56.85185
5 38.8 27.0 69.58763
9 55.5 31.7 57.11712
```

As a result all data have been selected where $dbh > 35$ cm. It is also possible to use more than one condition linked with a logical "and" (`&`):

```
 > myData[myData$dbh > 30 & myData$hd > 80, ]
    dbh    h       hd
1 33.5 28.9 86.26866
6 32.4 26.2 80.86420
8 31.0 25.5 82.25806
```

A logical "or" is expressed with the symbol `|`.

2.1.3 Input and output

Often the data we wish to process are too many to be inputted in R or are already available in digital format produced by another program. For the purpose of this book, it can be recommended to edit and to maintain data externally in MS Excel or in similar spreadsheet or database programs. The data to be analysed in R should include a header line and can then be conveniently converted to ASCII format. Every header should consist of a single word with continuous letters. Numeric and string values should not be mixed in a single column. If all data cells have valid entries it is easiest to convert the data to `text tabs delimited (*.txt)` in MS Excel. If some data cells are empty the conversion option `text comma delimited (*.csv)` should be preferred.

It is useful to start every new R script with the command `rm(list = ls())`, which removes all pre-existing objects and data. In line 2f. of the following code snippet the ASCII data saved in file `Data1.txt` are read by R and assigned to the data frame `myData`.

```
1  > rm(list = ls())
2  > myData <- read.table("/Cyhoeddi/SilvicultureBook/Data1.txt",
3  + header = T)
4  # myData <- read.csv("/Cyhoeddi/SilvicultureBook/Data1.csv",
5  # header=T)
6  > dim(myData)
7  [1] 56 6
8  > names(myData)
9  [1] "treeno" "species" "x" "y" "dbh" "h"
```

In lines 4f. the alternative code for comma delimited data is given. `header = T` indicates that the first row of the data file contains headers. The use of forward slashes is common practice in R irrespective of the operating system. The directory names are case sensitive and ideally should not contain blank spaces. When working in MS Windows a drive name, e.g. `C:` (without backslash) needs to be added immediately following the first quotation mark. It is also possible to substitute each forward slash by a double backslash (\\).

Ideally "." is always set as standard decimal separator in MS Excel, however, if commas were used the argument `dec = ","` separated by a comma in lines 3 or 5 will perform an automatic conversion.

The commands `dim` and `names` produce information on the dimensions of the new data frame and give the headers which simultaneously serve as vector names.

Missing data, which can originate from unmeasured tree characteristics, are represented in R by a logical constant with the value `NA` (**N**ot **A**vailable) and any operations on `NA` also yield `NA` as a result (Dalgaard, 2008, p. 14). In that case a special command can limit calculations to valid observations, i.e. missing values are then removed.

```
> mean(myData$dbh, na.rm = T)
```

`T` is short for `TRUE`. In R, it is possible to use both `T` or `TRUE` and `F` or `FALSE`, respectively. Alternatively records with no entry, i.e. with the value `NA`, can be removed beforehand using the following command.

```
> myData <- myData[!is.na(myData$dbh), ]
```

After modifying the data frame the data can be saved to ASCII files using one of the following commands. Naturally it is recommended to use a file name different from that of the input file to avoid overwriting.

```
> write.table(myData, file = "/Cyhoeddi/SilvicultureBook/
+ Data1new.txt")
> write.csv(myData, file = "/Cyhoeddi/SilvicultureBook/
+ Data1new.csv")
```

Subsets can be created in two different ways, by using (1) conditional statements in square brackets and (2) the `subset` function. In the following example lodgepole pine trees (*Pinus contorta* DOUGL. ex LOUD.) are selected for species specific analysis. In this context, the `length` function serves as a simple check up that both operations yield the same results. The comma towards the end of the first line is again responsible for selecting all data columns.

```
> myData.LP1 <- myData[myData$species == 3, ]
> length(myData.LP1$treeno)
[1] 18
> myData.LP2 <- subset(myData, species == 3)
> length(myData.LP2$treeno)
[1] 18
```

Similar to `sapply`, `tapply` allows the use of an implicit-loop command for exploring the number of trees per species. Again implicit loops are shortcuts to what otherwise would need to be programmed as long-winded loops.

```
> tapply(myData$treeno, myData$species, length)
 3 12
18 38
```

In this example, there are 18 lodgepole pine (*Pinus contorta* DOUGL. ex LOUD., coded as 3) and 38 Sitka spruce (*Picea sitchensis* (BONG.) CARR., coded as 12) trees in the data frame.

As we have already seen in the case of subsetting the same results can be obtained in R from more than one set of commands. In this case, the command `table(myData$species)` for example gives the same information.

2.1.4 Graphs and regressions

Graphs are essential elements of any data analysis. They visualise important relationships and help to spot mistakes. Graphs are a particular strength of R and convince with high quality that is only limited by the user's knowledge of how to program and to fine-tune them.

Following on with the data frame introduced in the last section, the diameter histogram is presented first. A histogram is a chart for displaying grouped continuous data, in which the width of each bar is proportional to the class interval and the area of each bar is proportional to the frequency it represents (Porkress, 2004, p. 118). All graphic commands are wrapped in the statements `pdf()` and `dev.off()`. This is a way to direct the graphical output to a pdf file for use in other computer programs such as LaTeX, MS PowerPoint or Apple Keynote. Other file formats are of course also possible, e.g. `png("*.png")`. If a screen output is intended lines 1 and 9 can be removed or disenabled with the `#` sign.

```
1  pdf(file = "/Cyhoeddi/SilvicultureBook/histExample.pdf")
2  > hist1 <- hist(myData$dbh, breaks = seq(8, 60, by = 4),
3  + include.lowest = T, right = F, plot = FALSE)
4  > hist1$counts <- hist1$counts / length(myData$dbh)
5  > par(mar = c(2, 4, 1, 2))
6  > plot(hist1, las = 1, cex.axis = 1.7, lwd = 2,
7  + ylim = c(0, 0.20), main = "", xlab = "", ylab = "")
8  > box(lwd = 2)
9  > dev.off()
10 null device
11          1
```

In principle the command `hist(myData$dbh)` is sufficient to produce a basic histogram. In the above code the histogram is, however, assigned to the variable `hist1` for further manipulation. The parameter `breaks = seq(8, 60, by = 4)` defines the diameter classes. The first two values give the minimum and maximum classes and can be established from the diameter range (`range(myData$dbh)`). The argument `by = 4` defines 4-cm diameter classes and the parameters `include.lowest = T` and `right = F` determine the forestry specific boundaries of diameter classes, see Chap. 1.2.

In line 4, the absolute frequencies in each diameter class are transformed to relative ones for better comparison with other forest stands. This is accomplished by dividing the absolute frequencies in each class by the total number of trees.

The plot margins are defined in line 5. From left to right the numbers represent the `bottom, left, top, right` margins of the plot and are interpreted in units of character widths (Jones *et al.*, 2009).

Finally the `plot` command in line 6f. commits R to produce the actual histogram using the modified histogram variable `hist1`. The argument `las = 1` organises the values on the ordinate in a horizontal way which is good for presentations and lectures. The arguments `cex.axis` and `lwd = 2` (= line width) defines the size of the axes labels and their width. `ylim` determines the extent of the abscissa, a similar command `xlim` also exists but is not necessary here. The histogram title and the axes labels can be determined with the arguments `main`, `xlab` and `ylab`, however, I personally prefer adding them with the `overpic` package of LaTeX for more flexibility and a better representation of mathematical symbols. Many more parameters exist to fine-tune histograms and the online help in R provides options and examples.

As a last step, a box with line width 2 has been added to the graph. Fig. 2.1 shows the result.

Bar charts are simple graphs that display single sets of numbers. In contrast to histograms the bars correspond to each number in the vector (Porkress, 2004).

To produce a bar chart of stem diameters a formula needs to be applied first to assign each stem diameter to a diameter class (line 1-2). Then the frequencies in each diameter class are established with the `tapply` function in line 3. Unfortunately, the

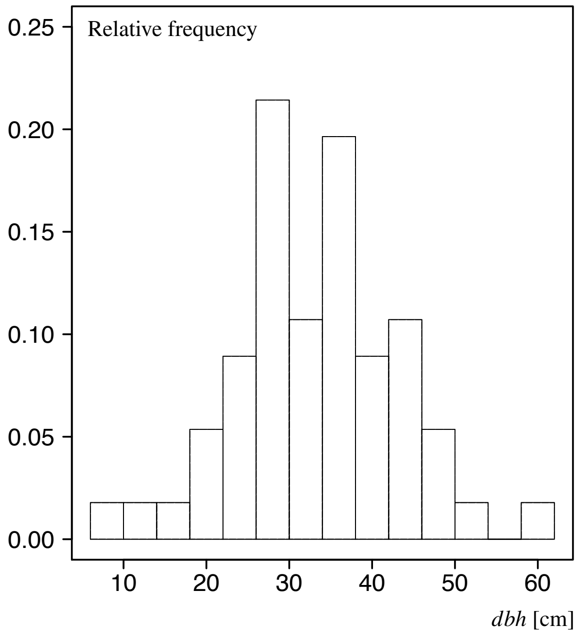

Fig. 2.1 Example of a diameter histogram based on a mixed Sitka spruce-lodgepole pine forest stand at Cefn Du (Clocaenog Forest, North Wales).

data now need to be re-formatted several times to fit the requirements of the bar plot function (lines 5-8). In line 9 and 10, relative frequencies are calculated by dividing the absolute frequencies by the number of observations. The class names are defined in line 11 as bar plot labels. Finally the bar plots are drawn in lines 12-17. Fig. 2.2 shows the result.

```
1  > myData$dbhcls <- 4 * (round((myData$dbh + 0.00000001) / 4) -
2  + 1) + 4
3  > dbhDistribution <- tapply(myData$dbh, myData$dbhcls, length)
4  > xdbhDistribution <- data.frame(cbind(dbhDistribution))
5  > xdbhDistribution$numbers <- xdbhDistribution$dbhDistribution
6  > xdbhDistribution$dbhcls <- as.numeric(row.names(
7  + xdbhDistribution))
8  > xdbhDistribution <- xdbhDistribution[c("dbhcls", "numbers")]
9  > tdbhDistribution <- xdbhDistribution$numbers /
10 + length(myData$dbh)
11 > names <- xdbhDistribution$dbhcls
12 > pdf(file = "/Cyhoeddi/SilvicultureBook/barplotExample.pdf")
13 > par(mar = c(3, 4, 3, 4))
14 > barplot(tdbhDistribution, beside = TRUE, ylab = "",
15 + main = "", las = 1, axes = TRUE, names.arg = names,
16 + cex.names = 1.7, cex.axis = 1.7, lwd = 2)
17 > dev.off()
18 null device
```

```
          1
```

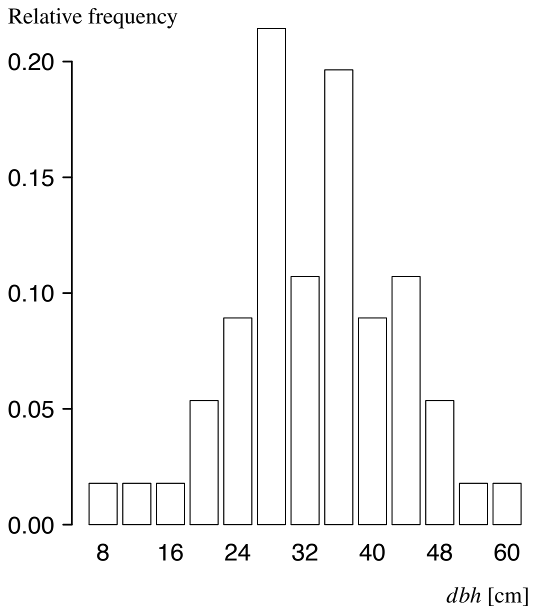

Fig. 2.2 Example of a diameter bar chart based on a mixed Sitka spruce-lodgepole pine forest stand at Cefn Du (Clocaenog Forest, North Wales).

Another important graph in data analysis is the scatterplot. We will apply this type of graph to visualise the relationship between stem diameters and total height in tree populations based on the same data frame as before.

```
> pdf(file = "Cyhoeddi/SilvicultureBook/scatterExample.pdf")
> par(mar = c(2, 3, 0.5, 0.5))
> plot(myData$dbh, myData$h, xlim = c(0, 65), ylim = c(0, 35),
+ pch = 16, xlab = "", ylab = "", axes = FALSE)
> axis(1, lwd = 2, cex.axis = 1.8)
> axis(2, las = 1, lwd = 2, cex.axis = 1.8)
> box(lwd = 2)
> dev.off()
null device
          1
```

Many parameters and arguments in the above code are already known from the histogram example. A scatterplot is produced using the `plot` function. First the abscissa values (stem diameter) are specified followed by the ordinate values (total tree height). The command `axes = FALSE` suppresses the drawing of the axes at this stage. This is performed separately by the following two axes commands. `pch = 16` defines the plotting characters as small filled circles. Other definitions are, of course, possible, `pch = 17` for example yields filled triangles.

It is now possible to define a trend curve and to display this curve in the scatterplot. This is a common application in data analysis and modelling. For this purpose we have to use nonlinear regression applied to a height-diameter model (see Chap.1.2). This can be achieved with the following code.

```
 > nls.Petterson <- nls(h ~ 1.3 + (dbh/(a + b * dbh))^3,
 + data = myData, start = list(a = 0.5, b = 3.8), trace = T)
28201.54 :  0.5 3.8
20851.01 :  33.8030885 -0.3563551
20756.15 :  33.6571018 -0.3536527
20567.28 :  33.3663958 -0.3482686
20192.88 :  32.790018 -0.337583
19456.83 :  31.657130 -0.316537
18031.2 :  29.4687529 -0.2757102
15335.76 :  25.3866524 -0.1988602
10422.76 :  18.3055084 -0.0627713
2790.814 :  8.035270 0.145125
838.3544 :  0.9757181 0.3110218
254.7894 :  1.8406893 0.2965715
238.5269 :  2.1459689 0.2887622
238.468 :  2.1665622 0.2882068
238.468 :  2.1671261 0.2881906
238.468 :  2.1671405 0.2881902
> summary(nls.Petterson)

Formula: h ~ 1.3 + (dbh/(a + b * dbh))^3

Parameters:
  Estimate Std. Error t value Pr(>|t|)
a 2.167141   0.184550   11.74   <2e-16 ***
b 0.288190   0.005458   52.80   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.101 on 54 degrees of freedom

Number of iterations to convergence: 15
Achieved convergence tolerance: 2.408e-07
```

In the first two lines, the model equation is specified followed by the data definition and the list of starting model parameters. `trace = T` confirms our intention to follow the individual iterations of the regression, which we can see in the following lines. Finally, the results are presented. Dalgaard (2008, p. 275ff.) provides a whole chapter with useful information on nonlinear regression in R.

The model parameters shown in the paragraph with the header "Parameters:" can now be used to superimpose the trend curve on the scatterplot. For this purpose we employ the `curve` function. This function allows the direct drawing of function graphs from equations.

```
 > curve(1.3 + (x/(summary(nls.Petterson)$coefficients[1] +
 + summary(nls.Petterson)$coefficients[2] * x))^3,
 + from = min(myData$dbh), to = max(myData$dbh),
 + lwd = 4, lty = 1, col = "grey", add = TRUE)
```

`summary(nls.Petterson)$coefficients[1]` gives the first and `summary(nls.Petterson)$coefficients[2]` the second regression coefficient. The rest of the code is quite self-explanatory. The parameter `add = TRUE` ensures that the scatterplot is not overwritten and `lty = 1` defines the line type, which in our case should be a simple continuous line. This code needs to be included before the command `dev.off()` after the regression. The final results can be seen in Fig. 2.3.

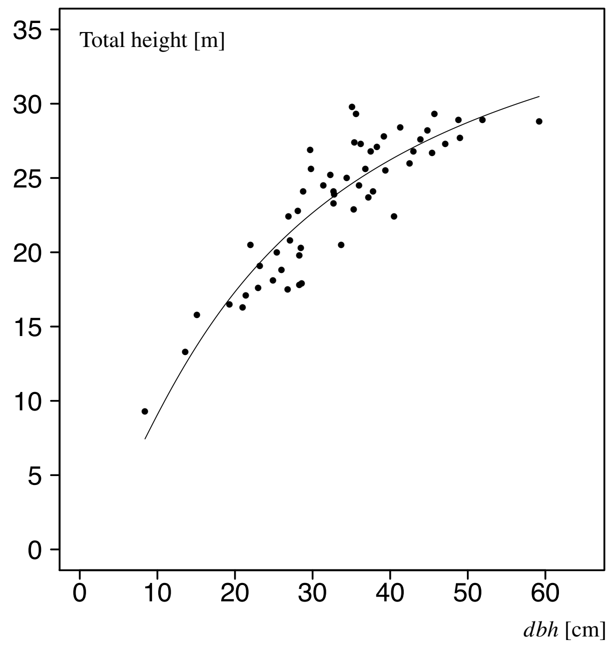

Fig. 2.3 Example of a scatterplot with a superimposed non-linear trend curve.

In some cases, often when more than two model parameters are involved, the `nls` routine stops processing or only produces results after several new attempts of devising new starting parameters, which can be awkward. In that case it can be recommended to make use of the `optim` function as outlined in Jones *et al.* (2009).

As part of this alternative to `nls` first a loss function needs to be defined, which is in lines 7-11 in the below code. In line 8, the Petterson function of lines 14-17 is called. The deviation between observed and estimated total tree heights is calculated in line 9 and returned to `optim` in line 10. In line 19, a vector with the starting values is defined.

```
1 > par(mar = c(2, 3, 0.5, 0.5))
2 > plot(myData$dbh, myData$h, xlim = c(0, 65), ylim = c(0, 35),
3  + pch = 16, xlab = "", ylab = "", axes = FALSE)
4  > axis(1, lwd = 2, cex.axis = 1.8)
5  > axis(2, las = 1, lwd = 2, cex.axis = 1.8)
6  >
7  > loss.L2 <- function(abdn, xdata) {
8  + yh <- xh(abdn[1 : 2], xdata)
9  + xdev <- xdata$h - yh
10 + return(sum(xdev^2, na.rm = TRUE))
11 + }
12 >
13 > # Petterson
14 > xh <- function(abdn, xdata) {
15 + yh <- 1.3 + (xdata$dbh/(abdn[1] + abdn[2] * xdata$dbh))^3
16 + return(yh)
17 + }
18 >
19 > abdn0 <- c(0.5, 3.8)
20 > abdn.L2 <- optim(abdn0, loss.L2, xdata = myData,
21 + control = list(maxit = 30000, temp = 2000,
22 + trace = TRUE, REPORT = 500))
23 >
24 > abdn.L2$par
25 [1] 2.1679653 0.2881626
26 >
27 > curve(1.3 + (x/(abdn.L2$par[1] + abdn.L2$par[2] * x))^3,
28 + from = min(myData$dbh), to = max(myData$dbh), lwd = 1,
29 + lty = 1, col = "black", add = TRUE)
30 >
31 > axis(1, lwd = 2, cex.axis = 1.8)
32 > axis(2, las = 1, lwd = 2, cex.axis = 1.8)
33 > box(lwd = 2)
```

Finally `optim` is run in lines 20-22 and the model parameters are produced by the command in line 24. The `curve` command in lines 27-29 adds the model curve to the plot.

Another common application of scatterplots is a situation where the user needs to draw curves which have been produced beforehand with other computer programs, e.g. in MS Excel. An example is shown in the code below and in Fig. 2.4. First, the external data defining the curves to be drawn need to be loaded (lines 2-3). Then the graphic functions and arguments are used that we already know from previous applications in this section. The main difference to usual scatterplots is that we now specify the graphical parameter `type = "l"`, which ensures that lines are drawn rather than points. In this example, the `plot()` function draws the mark variogram (Pommerening and Särkkä, 2013) as a solid curve.

```
1 > rm(list = ls())
2 > xfile <- read.table("Cyhoeddi/SilvicultureBook/
3 + Bialowieza.out", header = T)
4 > pdf(file = "Cyhoeddi/SilvicultureBook/BialowiezaGa.pdf")
5 > par(mar = c(2, 3.5, 0.5, 0.5))
6 > plot(xfile$r, xfile$ga, xlab="", ylab="", type = "l",
7 + las = 1, ylim = c(0,350), axes = FALSE, lwd = 2)
8 > lines(xfile$r, xfile$gamin, type = "l", col = "black",
+ lty = 3, lwd = 2)
> lines(xfile$r, xfile$gamax, type = "l", col = "black",
+ lty = 3, lwd = 2)
> axis(1, lwd = 2, cex.axis = 1.7)
> axis(2, las = 1, lwd = 2, cex.axis = 1.7)
> box(lwd = 2)
> dev.off()
null device
          1
```

Maximum and minimum variogram values from simulations are added as two additional curves using the `lines()` function, which otherwise have the same effect and graphical parameters as the `plot()` function. However, the graphical parameters `lty = 3` are used, specifying a dotted line type. The results can be seen in Fig. 2.4.

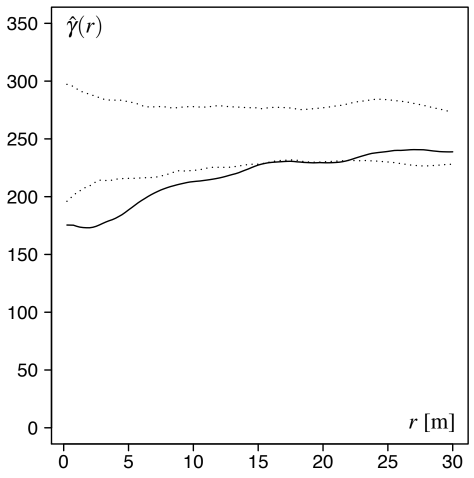

Fig. 2.4 Example of mark variogram curves, $\hat{\gamma}(r)$, produced from imported data in R. r is the intertree distance.

A box plot (or “box-and-whisker plot”) is an alternative to a histogram providing a visualisation of the main features of a data set (Braun and Murdoch, 2007, p. 36f.). The bold horizontal line marks the median and the upper and lower edges of the box give the upper and lower quartiles. About 50% of the data lies within the box, which defines the *interquartile range* (IQR). The dashed lines and end lines, the whiskers, lead to the smallest/largest value that is no smaller/no larger than 1.5 IQR below/above the lower/upper quartile. When data are drawn from the normal distribution or other distributions with a similar shape, about 99% of the observations fall between the whiskers. Outliers (observations that are very different from the rest of the data) are plotted as separate points (Braun and Murdoch, 2007, p. 37). Box plots are convenient for comparing distributions of data in two or more categories.

In the code example below, we compare the distributions of the *dbh* measurements between the two species, lodgepole pine (*Pinus contorta* DOUGL. ex LOUD.) and Sitka spruce (*Picea sitchensis* (BONG.) CARR.).

```
1  > my.labels <- c("LP", "SS")
2  > par(lab = c(length(my.labels), 5, 7), mar = c(2.5, 4, 0.5,
3  + 0.5))
4  > boxplot(dbh ~ species, data = myData, boxwex = 0.5, lwd = 2,
5  + axes = FALSE)
6  > axis(1, at = 1 : 2, labels = my.labels, lwd = 2,
7  + cex.axis = 1.8)
8  > axis(2, las = 1, lwd = 2, cex.axis = 1.8)
9  > box(lwd = 2)
```

Note that the abscissa labels are defined in line 1 and used in lines 2 and 6. `boxwex` is a scaling factor, which can be used to make the boxes narrower when there are only few groups. An additional argument `horizontal = TRUE` allows the drawing of horizontal box plots, which is sometimes useful. Fig. 2.5 shows the box plots produced from the above R code. They show that both species have quite different median diameters and diameter ranges. Sitka spruce ($n = 38$) has larger diameter trees and a wider range of tree sizes than lodgepole pine ($n = 18$).

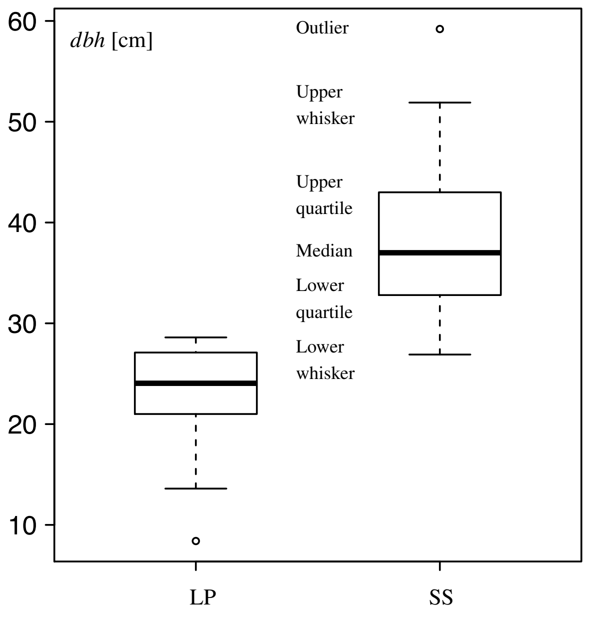

Fig. 2.5 Example of a box plot comparing two tree species of the same forest stand in terms of diameter distribution and the meaning of the box plot features.

In R, it is also possible to visualise a mathematical function directly without the need to compile a table of function values beforehand. This is very useful for visualising functions that we would like to consider as trend curves or models. As an example, we use here the Chapman-Richards growth function (Pienaar and Turnbull, 1973). In the following code, first the three function parameters are specified and then the graph is produced using the `curve` command.

```
# Specify parameters of Chapman-Richards growth function
> A <- 35.204
> k <- 0.0235
> p <- 1.237
> # Draw the curve of the growth function
> par(mar = c(4.5, 4.5, 2, 0.5))
> curve(A * (1 - exp(-k * x))^p, from = 5, to = 120,
+ xlab = "Age [years]", ylab = "Top height [m]",
+ main = "SP (YC 14, ITh, 1-8m)", axes = FALSE, lwd = 2)
> axis(1, lwd = 2, cex.axis = 1.8)
> axis(2, las = 1, lwd = 2, cex.axis = 1.8)
> box(lwd = 2)
```

This example of top height development over age is taken from the British yield table system (Hamilton and Christie, 1973), i.e. Scots pine (*Pinus sylvestris* L.), yield class 14, intermediate thinning and 1×0.5 m initial spacing. The result is given in Fig. 2.6, this time using the original R axes labels.

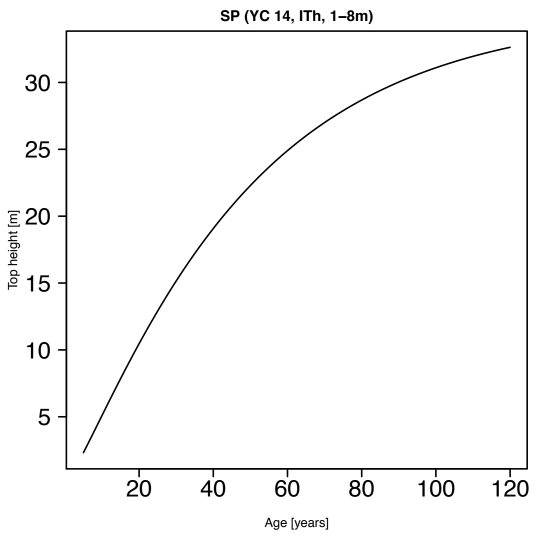

Fig. 2.6 Visualising the Chapman-Richards growth function.

As a summary the most important graph functions are listed in Table 2.3. Graph parameters can be found in the R online help. Incidentally several graphs can be simultaneously displayed on screen or in a file using the graphical parameters `par(mfrow = c(1, 2))`. In this case two graphs are shown in one row and two columns.

Table 2.3 Important graph functions.

Function	Description
`plot()`	Scatterplot (and more)
`hist()`	Histogram
`barplot()`	Barplot graph
`boxplot()`	Boxplot graph
`lorenz()`	Lorenz curve
`pie()`	Pie chart

Just for completeness let us also consider the case of a simple linear regression. For this purpose we will work with the following two vectors of stem diameters in cm and the corresponding diameter increments also in cm:

```
> dbh <- c(42.4, 54.0, 32.2, 39.8, 26.1, 30.4, 39.9, 45.5,
+ 37.9, 31.7)
> id <- c(5.5, 7.2, 4.0, 5.1, 3.1, 3.8, 5.1, 6.0, 4.8, 4.0)
```

These are sample data from a Sitka spruce (*Picea sitchensis* (Bong.) Carr.) forest stand at Clocaenog forest in North Wales. The cycle of repeated *dbh* measurements that form the basis of the increment values is 5 years.

A simple linear regression between the initial stem diameters, `dbh`, i.e. the diameters at the beginning of the increment period, and the diameter increment, `id`, as independent variable can be performed with the following code.

```
> lm.inc <- lm(id ~ dbh)
> summary(lm.inc)
Call:
lm(formula = id ~ dbh)

Residuals:
      Min        1Q    Median        3Q       Max
-0.046878 -0.025837 -0.009401  0.035255  0.057057

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.678790   0.062513  -10.86 4.57e-06 ***
dbh          0.145796   0.001612   90.46 2.49e-13 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.03984 on 8 degrees of freedom
Multiple R-squared: 0.999,Adjusted R-squared: 0.9989
F-statistic:  8183 on 1 and 8 DF,  p-value: 2.489e-13
```

Line 1 gives the code of the linear regression and the following lines are the result of the `summary()` command.

Incidentally, intercept (`a`), slope (`b`) parameters and coefficient of determination, R^2, can also be calculated “manually” in the following way:

```
> (b <- cov(dbh, id) / var(dbh))
[1] 0.145796
> (a <- mean(id) - b * mean(dbh))
[1] -0.6787903
> cor(dbh, id)^2
[1] 0.9990233
```

We can now use the regression results again to plot a trend line. Consider the following code.

```
 par(mar = c(2, 2, 0.5, 0.8))
> plot(dbh, id, ylab = "", xlab = "", pch = 16, cex = 1.7,
+ las = 1, cex.axis = 1.7, lwd = 2)
> lines(dbh, fitted(lm.inc), col = "black")
> box(lwd = 2)
```

The `lines()` command adds the actual regression line as shown in Fig. 2.7. The command `abline(lm.inc, col = "black")` produces a similar trend line.

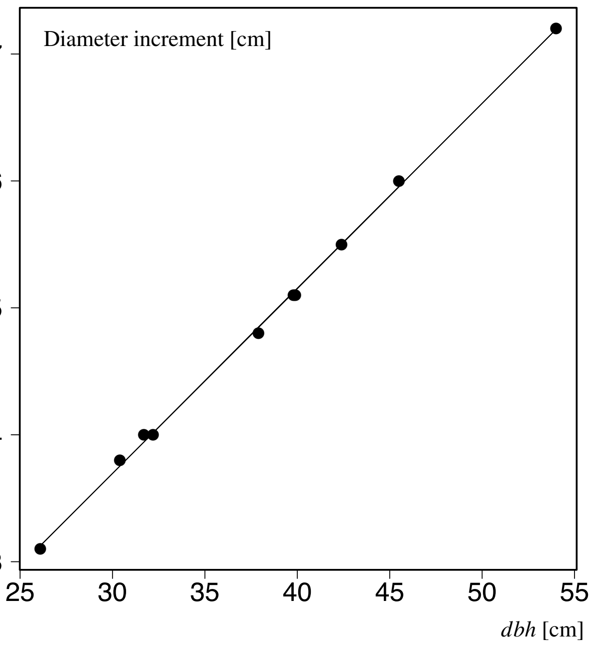

Fig. 2.7 Example of a scatterplot with a superimposed linear trend curve.

For examining the normality of the residuals graphically we can for example employ the QQ plot (quantile quantile plot). It draws the sample quantiles against the quantiles of a normal distribution, see Fig. 2.8.

```
> qqnorm(lm.inc$residuals)
> qqline(lm.inc$residuals)
```

Without going into detail, the data points should ideally lie on a straight line. In our case, there are only very few observations to make this a worthwhile exercise.

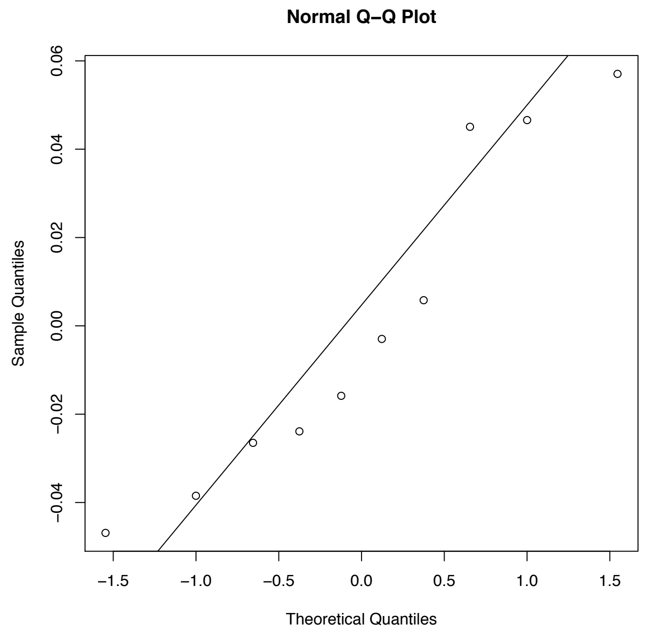

Fig. 2.8 Normal quantile plot of residuals.

Incidentally, it sometimes makes sense to assume that a regression line passes through the origin, `(0, 0)`, i.e. the intercept of the regression line is zero. This can be specified in the model formula by adding the term `-1` (“minus intercept”) on the right-hand side Dalgaard (2008, p. 198): `id ~ dbh - 1`. Again Dalgaard (2008, p. 109ff.) has prepared a chapter on simple linear regression in R, which is recommended for further reading.

2.1.5 Functions and flow control

`min()`, `max()` and `sum()` are functions that are part of the standard libraries of R. The user has, of course, also the possibility to define his own function. To write functions is a way to reduce complexity by writing only once certain elements of code that are frequently used. Functions are self-contained units. They take inputs, do calculations and produce outputs (Braun and Murdoch, 2007). As an example we use the calculation of the quadratic mean diameter (see Chap. 1.2).

The syntax is very simple. The word `function` followed by the list of arguments in round brackets, `()`, is assigned to the function name.

```
> calcQuadraticMean <- function(myVector) {
+   x <- sqrt(sum(myVector^2) / length(myVector))
+   return(x)
+ }
```

In a next step, the quadratic mean is calculated and assigned to the variable *x*. The value of *x* is then finally returned to the calling instance with the statement `return(x)`. The whole function body is wrapped in curly braces, `{}`. The location of the first, opening curly brace can be at the end of the first line or at the beginning of the second, this is a matter of taste.

Prior to using the new function you need once to highlight the whole function code and to execute the `run` command (`Ctrl-R` or `Cmd-Enter`). Possible errors in the code will be identified by R at this stage. Afterwards you can apply your own function in all possible contexts, e.g.

```
> calcQuadraticMean(myData$dbh)
[1] 34.7506
```

Since the calculations in the function `calcQuadraticMean` are simple and consist only of a single line, the function code can be simplified to

```
> calcQuadraticMean <- function(myVector)
+   return(sqrt(sum(myVector^2) / length(myVector)))
```

However, it is important that every user adopts a programming style that best works for him. Above all any code has to be correct in the first place.

Structures of flow control are useful for repetitive tasks. Often such tasks can be avoided by pre-defined functions and implicit loops in R. In case they are really necessary for a given task a brief introduction is given here by continuing the work with the data frame used in this section and the `mean` function. Applied to the data set that we have imported earlier from an ASCII file the following code can be employed.

```
> mean(myData$dbh)
[1] 33.34107
```

Since an R function is available there is no need for programming the calculation of arithmetic means from scratch. However, expressing the same calculation by means of loops helps to understand how they work and how they can be implemented. The following listing shows how the `mean` function can be simulated with a `for` loop.

```
1  > mean.xfor <- 0
2  > for (i in 1 : length(myData$dbh)) {
3  +  mean.xfor <- mean.xfor + myData$dbh[i]
4  + }
5  > mean.xfor <- mean.xfor / length(myData$dbh)
6  > mean.xfor
7  [1] 33.34107
```

In line 1, the variable collecting the summed stem diameter values is set to zero for initialisation. The loop statement defining the beginning and the end of the loop is given in line 2. In line 3, all stem diameter values are summed up. Note that individual *dbh* values are accessed by putting the running variable *i* in square brackets. The loop is executed exactly `length(myData$dbh)` times. Finally the sum is divided by the number of observations.

Since only one line of code is included in the `for` loop the above listing can be simplified by omitting the curly braces. Also the whole code can be substituted by the simpler statement `sum(myData$dbh) / length(myData$dbh)`.

The arithmetic mean can also be implemented with a `while` loop. A little more effort is required as the running variable *i* needs to be initialised (line 2) and incremented (line 5). The `while` loop is particularly useful where the number of repetitions is difficult to determine beforehand.

```
1  > mean.xwhile <- 0
2  > i <- 1
3  > while (i < length(myData$dbh) + 1) {
4  +  mean.xwhile <- mean.xwhile + myData$dbh[i]
5  +  i <- i + 1
6  + }
7  > mean.xwhile <- mean.xwhile / (i - 1)
8  > mean.xwhile
9  [1] 33.34107
```

Another option in R is the `repeat` loop, where the abort condition is at the bottom of the loop introduced by an `if` statement.

```
1  > mean.xrepeat <- 0
2  > i <- 1
3  > repeat {
4  +  mean.xrepeat <- mean.xrepeat + myData$dbh[i]
5  +  i <- i + 1
6  +  if(i == length(myData$dbh) + 1)
7  +  break
8  + }
9  > mean.xrepeat <- mean.xrepeat / (i - 1)
10 > mean.xrepeat
11 [1] 33.34107
```

The `repeat` loop is not frequently used. However, the `if` statement is quite common, mostly in connection with loops and should also include an `else` statement for alternative pathways in programs.

Many loops can be and should be avoided in R as they require a lot of computation time. In the following code snippet one of three height bands is assigned to each tree depending on the maximum tree height in a forest stand `maxh` and the total height `myData$ht` of each tree.

```
> myData$band <- 1
> myData$band[myData$ht > max(myData$ht) * 0.8] <- 3
> myData$band[myData$ht > max(myData$ht) * 0.5 &
+ myData$ht <= max(myData$ht) * 0.8] <- 2
```

Instead of using a `for` loop and `if` statements it is possible here to exploit the strengths of R. In this case, a default value of 1 is assigned to all trees and then the two other values are assigned using relational statements in square brackets. This strategy makes use of the vector philosophy of R and the relational statements replace the traditional `if` statements.

2.1.6 Extending R with C++

In some situations, loops are unavoidable and in that case it is recommended to program them externally in C++ or Fortran. R can then link to this compiled or uncompiled code. This is a kind of 'outsourcing', i.e. the strengths of both R and higher programming languages are combined. Complex loops run much faster in C++ or Fortran and considerable time savings can be achieved with this strategy.

A simple way of linking C++ to R is using for example the Rcpp package (Eddelbuettel and François, 2011; Eddelbuettel, 2013). This package requires R versions > 2.15.2. Using the R Package Installer the package Rcpp first needs to be installed in R. When working with Mac OS Xcode needs to be installed from the Apple Developer site (`https://developer.apple.com/xcode/`). On MS Windows you need to download and install Rtools (`http://cran.r-project.org/bin/windows /Rtools/`). Problems, however, can occur in MS Windows 7/8. You need to make sure that the file path leading to R does not contain empty space, e.g. `C:\Program Files\ ...`.

To illustrate the use of combined C++ and R code we continue with the theoretical example of the previous section. In a higher programming language, calculating an arithmetic mean usually involves using a loop. Therefore we program this part in C++ and the corresponding code is given below and saved as meanVector.cpp in ASCII format.

```
#include <Rcpp.h>

using namespace Rcpp;

// [[Rcpp::export]]
double meanVector(NumericVector a) {

   int n = a.size();
   double xsum = 0;

   for (int i = 0; i < n; i++)
12          xsum += a[i];
13
14    return xsum / n;
15 }
```

In lines 1 and 3 reference to the Rcpp package is made. The code in line 5 is necessary in order to be able to access the function `meanVector` from within R. The algorithmic structure is essentially the same that we already know from the previous section, only the programming language and the syntax that comes with it is different. Also the Rcpp package provides a number of variable types that otherwise do not exist in C++, e.g. `NumericVector` in line 6. These additional types allow a smoother cooperation with R.

The C++ function `meanVector` is now ready for use in R. In line 1 of the code below all pre-existing R objects are deleted from memory. Then the Rcpp library is loaded in line 3. The code in line 5 is crucial, as the C++ code is loaded and compiled "on the fly". This may take a few seconds. Potential C++ programming errors can now be viewed in the R console.

```
1  > rm(list = ls())
2  >
3  > library(Rcpp)
4  >
5  > sourceCpp("/Users/arnepommerening/Dropbox/Rcpp/
6  + meanVector.cpp")
7  > v <- c(1.5, 2.9, 3.2, 4.1, 5.5)
8  > mean(v)
9  [1] 3.44
10 > meanVector(v)
11 [1] 3.44
```

If the compilation was successful it is now possible to use the C++ function `meanVector` from within R. For this purpose we first construct a vector `v` in line 7. Then we use the R function `mean` to compute the arithmetic mean of the vector values. In a final step, the self-made function `meanVector` is applied to calculate the same mean of vector values in line 10 and the results in lines 9 and 11 coincide.

In analogy to this simple example more complex (simulation) code can be programmed in C++ and afterwards called from within R. It is for example possible to implement a whole individual tree model in C++/R. For this purpose all detailed core model functions are implemented in C++ and then called in R where the simulation results are then statistically analysed and visualised in graphs.

For details of the C and C++ languages please refer to the books and manuals on that language, which exist in great abundance. Also note that the syntax of Java and C are quite similar so that code can easily and quickly be modified for use in Rcpp/R.