An introduction to R

Longhow Lam

Under Construction Jan-2010 some sections are unfinished!

longhowlam at gmail dot com

Contents

1 Introduction **7**
- 1.1 What is R? . . . 7
- 1.2 The R environment . . . 8
- 1.3 Obtaining and installing R . . . 8
- 1.4 Your first R session . . . 9
- 1.5 The available help . . . 11
  - 1.5.1 The on line help . . . 11
  - 1.5.2 The R mailing lists and the R Journal . . . 12
- 1.6 The R workspace, managing objects . . . 12
- 1.7 R Packages . . . 13
- 1.8 Conflicting objects . . . 15
- 1.9 Editors for R scripts . . . 16
  - 1.9.1 The editor in RGui . . . 16
  - 1.9.2 Other editors . . . 16

2 Data Objects **19**
- 2.1 Data types . . . 19
  - 2.1.1 Double . . . 19
  - 2.1.2 Integer . . . 20
  - 2.1.3 Complex . . . 21
  - 2.1.4 Logical . . . 21
  - 2.1.5 Character . . . 22
  - 2.1.6 Factor . . . 23
  - 2.1.7 Dates and Times . . . 25
  - 2.1.8 Missing data and Infinite values . . . 27
- 2.2 Data structures . . . 28
  - 2.2.1 Vectors . . . 28
  - 2.2.2 Matrices . . . 32
  - 2.2.3 Arrays . . . 34
  - 2.2.4 Data frames . . . 35
  - 2.2.5 Time-series objects . . . 37
  - 2.2.6 Lists . . . 38
  - 2.2.7 The `str` function . . . 41

3 Importing data **42**
- 3.1 Text files . . . 42

3.1.1 The `scan` function . . . 44
3.2 Excel files . . . 44
3.3 Databases . . . 45
3.4 The Foreign package . . . 46

4 Data Manipulation **47**
4.1 Vector subscripts . . . 47
4.2 Matrix subscripts . . . 51
4.3 Manipulating Data frames . . . 54
4.3.1 Extracting data from data frames . . . 54
4.3.2 Adding columns to a data frame . . . 57
4.3.3 Combining data frames . . . 57
4.3.4 Merging data frames . . . 59
4.3.5 Aggregating data frames . . . 60
4.3.6 Stacking columns of data frames . . . 61
4.3.7 Reshaping data . . . 61
4.4 Attributes . . . 62
4.5 Character manipulation . . . 63
4.5.1 The functions `nchar`, `substring` and `paste` . . . 64
4.5.2 Finding patterns in character objects . . . 65
4.5.3 Replacing characters . . . 67
4.5.4 Splitting characters . . . 68
4.6 Creating factors from continuous data . . . 68

5 Writing functions **70**
5.1 Introduction . . . 70
5.2 Arguments and variables . . . 72
5.2.1 Required and optional arguments . . . 72
5.2.2 The '`...`' argument . . . 73
5.2.3 Local variables . . . 73
5.2.4 Returning an object . . . 74
5.2.5 The Scoping rules . . . 75
5.2.6 Lazy evaluation . . . 76
5.3 Control flow . . . 77
5.3.1 Tests with `if` and `switch` . . . 77
5.3.2 Looping with `for`, `while` and `repeat` . . . 79
5.4 Debugging your R functions . . . 80
5.4.1 The `traceback` function . . . 80
5.4.2 The `warning` and `stop` functions . . . 81
5.4.3 Stepping through a function . . . 82
5.4.4 The `browser` function . . . 83

6 Efficient calculations **84**
6.1 Vectorized computations . . . 84

6.2 The apply and outer functions . . . 86
6.2.1 the `apply` function . . . 86
6.2.2 the `lapply` and `sapply` functions . . . 87
6.2.3 The `tapply` function . . . 89
6.2.4 The `by` function . . . 90
6.2.5 The `outer` function . . . 91
6.3 Using Compiled code . . . 92
6.3.1 The `.C` and `.Fortran` interfaces . . . 93
6.3.2 The `.Call` and `.External` interfaces . . . 94
6.4 Some Compiled Code examples . . . 94
6.4.1 The arsim example . . . 94
6.4.2 Using `#include <R.h>` . . . 96
6.4.3 Evaluating R expressions in C . . . 98

7 Graphics **103**
7.1 Introduction . . . 103
7.2 More plot functions . . . 104
7.2.1 The `plot` function . . . 105
7.2.2 Distribution plots . . . 106
7.2.3 Two or more variables . . . 109
7.2.4 Graphical Devices . . . 111
7.3 Modifying a graph . . . 112
7.3.1 Graphical parameters . . . 112
7.3.2 Some handy low-level functions . . . 119
7.3.3 Controlling the axes . . . 121
7.4 Trellis Graphics . . . 123
7.4.1 Introduction . . . 123
7.4.2 Multi panel graphs . . . 125
7.4.3 Trellis panel functions . . . 128
7.4.4 Conditioning plots . . . 130
7.5 The ggplot2 package . . . 131
7.5.1 The `qplot` function . . . 131
7.5.2 Facetting . . . 133
7.5.3 Plots with several layers . . . 134

8 Statistics **135**
8.1 Basic statistical functions . . . 135
8.1.1 Statistical summaries and tests . . . 135
8.1.2 Probability distributions and random numbers . . . 139
8.2 Regression models . . . 141
8.2.1 Formula objects . . . 141
8.3 Linear regression models . . . 142
8.3.1 Formula objects . . . 142
8.3.2 Modeling functions . . . 144

8.3.3 Multicollinearity . . . 149
8.3.4 Factor (categorical) variables as regression variables . . . 152
8.4 Logistic regression . . . 154
8.4.1 The modeling function `glm` . . . 155
8.4.2 Performance measures . . . 157
8.4.3 Predictive ability of a logistic regression . . . 158
8.5 Tree models . . . 160
8.5.1 An example of a tree model . . . 161
8.5.2 Coarse classification and binning . . . 162
8.6 Survival analysis . . . 164
8.6.1 The Cox proportional hazards model . . . 165
8.6.2 Parametric models for survival analysis . . . 169
8.7 Non linear regression . . . 170
8.7.1 Ill-conditioned models . . . 174
8.7.2 Singular value decomposition . . . 177

9 Miscellaneous Stuff 180
9.1 Object Oriented Programming . . . 180
9.1.1 Introduction . . . 180
9.1.2 Old style classes . . . 180
9.1.3 New Style classes . . . 184
9.2 R Language objects . . . 189
9.2.1 Calls and Expressions . . . 189
9.2.2 Expressions as Lists . . . 190
9.2.3 Functions as lists . . . 192
9.3 Calling R from SAS . . . 193
9.3.1 The `call system` and `X` functions . . . 193
9.3.2 Using SAS data sets and SAS ODS . . . 194
9.4 Defaults and preferences in R, Starting R, . . . 196
9.4.1 Defaults and preferences . . . 196
9.4.2 Starting R . . . 197
9.5 Creating an R package . . . 198
9.5.1 A 'private' package . . . 198
9.5.2 A 'real' R package . . . 199
9.6 Calling R from Java . . . 202
9.7 Creating fancy output and reports . . . 204
9.7.1 A simple LaTeX-table . . . 205
9.7.2 An simple HTML report . . . 206

Bibliography 207

Index 208

List of Figures

1.1 The R system on Windows . . . 9
1.2 R integrated in the Eclipse development environment . . . 17
1.3 The Tinn-R and an the R Console environment . . . 18

6.1 A surface plot created with the function `persp` . . . 92
6.2 Calculation times of `arsimR` (solid line) and `arsimC` (dashed line) for increasing vectors . . . 97

7.1 A scatterplot with a title . . . 104
7.2 Line plot with title, can be created with `type="l"` or the `curve` function. 105
7.3 Different uses of the function `plot` . . . 106
7.4 Example distribution plot in R . . . 107
7.5 Example barplot where the first argument is a matrix . . . 108
7.6 Example graphs of multi dimensional data sets . . . 110
7.7 The different regions of a plot . . . 114
7.8 The plotting area of this graph is divided with the `layout` function. . . . 116
7.9 Examples of different symbols and colors in plots . . . 119
7.10 The graph that results from the previous low-level plot functions. . . . 121
7.11 Graphs resulting from previous code examples of customizing axes. . . . 124
7.12 Trellis plot Price versus Weight for different types . . . 126
7.13 A trellis plot with two conditioning variables . . . 127
7.14 Histogram of mileage for different weight classes . . . 128
7.15 Trellis plot with modified panel function . . . 130
7.16 Trellis plot adding a least squares line in each panel . . . 131
7.17 A coplot with two conditioning variables . . . 132
7.18 A coplot with a smoothing line . . . 133

8.1 A histogram and a qq-plot of the model residuals to check normality of the residuals. . . . 148
8.2 Diagnostic plots to check for linearity and for outliers. . . . 149
8.3 Explorative plots giving a first impression of the relation between the binary `y` variable and `x` variables. . . . 156
8.4 The ROC curve to assess the quality of a logistic regression model . . . 159
8.5 Plot of the tree: Type is predicted based on Mileage and Price . . . 162
8.6 Binning the age variable, two intervals in this case . . . 164

8.7 Survival curve: 10% will develop AIDS before 45 months and 20% before 76 months. . . . 166
8.8 Scatter plot of the martingale residuals . . . 168
8.9 Three subjects with age 10, 30 and 60 . . . 169
8.10 Scatter plot of our simulated data for `nls` . . . 172
8.11 Simulated data and `nls` predictions . . . 175
8.12 Hill curves for two sets of parameters . . . 176

9.1 Result of the specific plot method for class bigMatrix. . . . 185
9.2 Some Lissajous plots . . . 203
9.3 A small java gui that can call R functions. . . . 204

1 Introduction

1.1 What is R?

While the commercial implementation of S, S-PLUS, is struggling to keep its existing users, the open source version of S, R, has received a lot of attention in the last five years. Not only because the R system is a free tool, the system has proven to be a very effective tool in data manipulation, data analysis, graphing and developing new functionality. The user community has grown enormously the last years, and it is an active user community writing new R packages that are made available to others.

If you have any questions or comments on this document please do not hesitate to contact me.

The best explanation of R is given on the R web site http://www.r-project.org. The remainder of this section and the following section are taken from the R web site.

R is a language and environment for statistical computing and graphics. It is a GNU project which is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R.

R provides a wide variety of statistical (linear and non linear modeling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible. The S language is often the vehicle of choice for research in statistical methodology, and R provides an Open Source route to participation in that activity.

One of R's strengths is the ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control.

R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License in source code form. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS.

1.2 The R environment

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or on hardcopy, and
- a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

The term 'environment' is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.

R, like S, is designed around a true computer language, and it allows users to add additional functionality by defining new functions. Much of the system is itself written in the R dialect of S, which makes it easy for users to follow the algorithmic choices made. For computationally-intensive tasks, C, C++ and Fortran code can be linked and called at run time. Advanced users can write C code to manipulate R objects directly.

Many users think of R as a statistics system. We prefer to think of it of an environment within which statistical techniques are implemented. R can be extended (easily) via packages. There are about eight packages supplied with the R distribution and many more are available through the CRAN family of Internet sites covering a very wide range of modern statistics.

R has its own LaTeX-like documentation format, which is used to supply comprehensive documentation, both on-line in a number of formats and in hardcopy.

1.3 Obtaining and installing R

R can be downloaded from the 'Comprehensive R Archive Network' (CRAN). You can download the complete source code of R, but more likely as a beginning R user you want to download the precompiled binary distribution of R. Go to the R web site http://www.r-project.org and select a CRAN mirror site and download the base distribution file, under Windows: `R-2.7.0-win32.exe`. At the time of writing the latest version is 2.7.0. We will mention user contributed packages in the next section.

The base file has a size of around 29MB, which you can execute to install R. The installation wizard will guide you through the installation process. It may be useful to

install the R reference manual as well, by default it is not installed. You can select it in the installation wizard.

1.4 Your first R session

Start the R system, the main window (RGui) with a sub window (R Console) will appear as in figure 1.1.

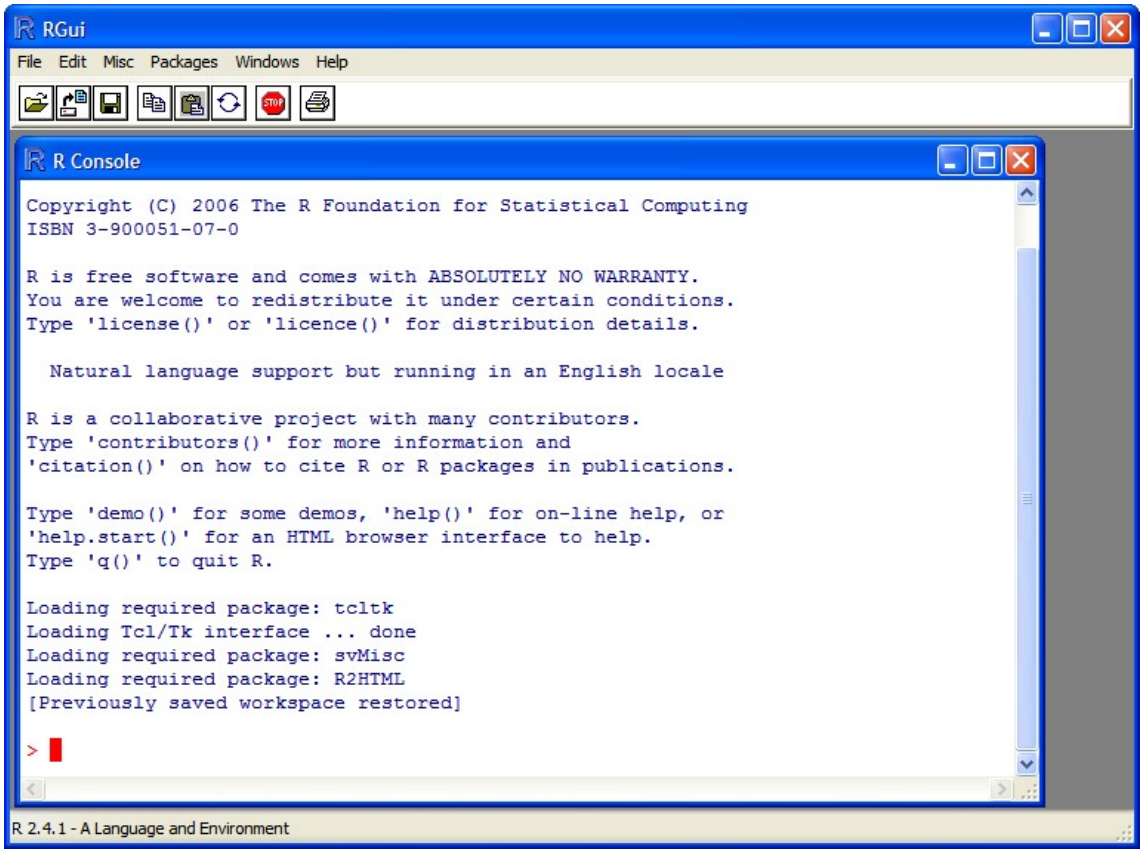

Figure 1.1: The R system on Windows

In the 'Console' window the cursor is waiting for you to type in some R commands. For example, use R as a simple calculator:

```
> print("Hello world!")
[1] "Hello world!"
> 1 + sin(9)
[1] 1.412118
> 234/87754
[1] 0.002666545
```

```
> (1 + 0.05)^8
[1] 1.477455
> 23.76*log(8)/(23 + atan(9))
[1] 2.019920
```

Results of calculations can be stored in objects using the assignment operators:

- An arrow (<-) formed by a smaller than character and a hyphen without a space!
- The equal character (=).

These objects can then be used in other calculations. To print the object just enter the name of the object. There are some restrictions when giving an object a name:

- Object names cannot contain 'strange' symbols like !, +, -, #.
- A dot (.) and an underscore (_) are allowed, also a name starting with a dot.
- Object names can contain a number but cannot start with a number.
- R is case sensitive, `X` and `x` are two different objects, as well as `temp` and `temP`.

```
> x = sin(9)/75
> y = log(x) + x^2
> x
[1] 0.005494913
> y
[1] -5.203902
> m <- matrix(c(1,2,4,1), ncol=2)
> m
>      [,1] [,2]
[1,]    1    4
[2,]    2    1
> solve(m)
           [,1]       [,2]
[1,] -0.1428571  0.5714286
[2,]  0.2857143 -0.1428571
```

To list the objects that you have in your current R session use the function `ls` or the function `objects`.

```
> ls()
[1] "x" "y"
```

So to run the function `ls` we need to enter the name followed by an opening (and and a closing). Entering only `ls` will just print the object, you will see the underlying R code of the the function `ls`. Most functions in R accept certain arguments. For example, one of the arguments of the function `ls` is `pattern`. To list all objects starting with the letter x:

```
> x2 = 9
> y2 = 10
> ls(pattern="x")
[1] "x"  "x2"
```

If you assign a value to an object that already exists then the contents of the object will be overwritten with the new value (without a warning!). Use the function `rm` to remove one or more objects from your session.

```
> rm(x, x2)
```

To conclude your first session, we create two small vectors with data and a scatterplot.

```
z2 <- c(1,2,3,4,5,6)
z3 <- c(6,8,3,5,7,1)
plot(z2,z3)
title("My first scatterplot")
```

After this very short R session which barely scratched the surface, we hope you continue using the R system. The following chapters of this document will explain in detail the different data types, data structures, functions, plots and data analysis in R.

1.5 The available help

1.5.1 The on line help

There is extensive on line help in the R system, the best starting point is to run the function `help.start()`. This will launch a local page inside your browser with links to the R manuals, R FAQ, a search engine and other links.

In the R Console the function `help` can be used to see the help file of a specific function.

```
help(mean)
```

Use the function `help.search` to list help files that contain a certain string.

```
> help.search("robust")
Help files with alias or concept or title matching 'robust' using fuzzy
matching:

hubers(MASS)            Huber Proposal 2 Robust Estimator of Location
                        and/or Scale
rlm(MASS)               Robust Fitting of Linear Models
summary.rlm(MASS)       Summary Method for Robust Linear Models
line(stats)             Robust Line Fitting
runmed(stats)           Running Medians -- Robust Scatter Plot
                        Smoothing

Type 'help(FOO, package = PKG)' to inspect entry 'FOO(PKG) TITLE'.
```

The R manuals are also on line available in pdf format. In the RGui window go the help menu and select 'manuals in pdf'.

1.5.2 The R mailing lists and the R Journal

There are several mailing lists on R, see the R website. The main mailing list is R-help, web interfaces are available where you can browse trough the postings or search for a specific key word. If you have a connection to the internet, then the function `RSiteSearch` in R can be used to search for a string in the archives of all the R mailing lists.

```
RSiteSearch("MySQL")
```

Another very useful webpage on the internet is www.Rseek.org, a sort of R search enigine. Also take a look at the R Journal, at http://journal.r-project.org.

1.6 The R workspace, managing objects

Objects that you create during an R session are hold in memory, the collection of objects that you currently have is called the workspace. This workspace is not saved on disk unless you tell R to do so. This means that your objects are lost when you close R and not save the objects, or worse when R or your system crashes on you during a session.

When you close the RGui or the R console window, the system will ask if you want to save the workspace image. If you select to save the workspace image then all the objects

in your current R session are saved in a file `.RData`. This is a binary file located in the working directory of R, which is by default the installation directory of R.

During your R session you can also explicitly save the workspace image. Go to the 'File' menu and then select 'Save Workspace...', or use the `save.image` function.

```
## save to the current working directory
save.image()
## just checking what the current working directory is
getwd()

## save to a specific file and location
save.image("C:\\Program Files\\R\\R-2.5.0\\bin\\.RData")
```

If you have saved a workspace image and you start R the next time, it will restore the workspace. So all your previously saved objects are available again. You can also explicitly load a saved workspace file, that could be the workspace image of someone else. Go the 'File' menu and select 'Load workspace...'.

1.7 R Packages

One of the strengths of R is that the system can easily be extended. The system allows you to write new functions and package those functions in a so called 'R package' (or 'R library'). The R package may also contain other R objects, for example data sets or documentation. There is a lively R user community and many R packages have been written and made available on CRAN for other users. Just a few examples, there are packages for portfolio optimization, drawing maps, exporting objects to html, time series analysis, spatial statistics and the list goes on and on. In section 9.5.1 we'll give a short description on writing your own package.

When you download R, already a number (around 30) of packages are downloaded as well. To use a function in an R package, that package has to be attached to the system. When you start R not all of the downloaded packages are attached, only seven packages are attached to the system by default. You can use the function `search` to see a list of packages that are currently attached to the system, this list is also called the *search path*.

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

The first element of the output of `search` is `".GlobalEnv"`, which is the current workspace of the user. To attach another package to the system you can use the menu or the `library` function. Via the menu: Select the 'Packages' menu and select 'Load package...', a list of available packages on your system will be displayed. Select one and click 'OK', the package is now attached to your current R session. Via the `library` function:

```
> library(MASS)
> shoes
$A
 [1] 13.2  8.2 10.9 14.3 10.7  6.6  9.5 10.8  8.8 13.3

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

The function `library` can also be used to list all the available libraries on your system with a short description. Run the function without any arguments

```
> library()
Packages in library 'C:/PROGRA~1/R/R-25~1.0/library':

base                    The R Base Package
boot                    Bootstrap R (S-Plus) Functions (Canty)
class                   Functions for Classification
cluster                 Cluster Analysis Extended Rousseeuw et al.
codetools               Code Analysis Tools for R
datasets                The R Datasets Package
DBI                     R Database Interface
foreign                 Read Data Stored by Minitab, S, SAS, SPSS,
                        Stata, Systat, dBase, ...
graphics                The R Graphics Package
...
...
```

If you have a connection to the internet then a package on CRAN can be installed very easily. To install a new package go to the 'Packages' menu and select 'Install package(s)...'. Then select a CRAN mirror near you, a (long) list with all the packages will appear where you can select one or more packages. Click 'OK' to install the selected packages. Note that the packages are only installed on your machine and not loaded (attached) to your current R session. As an alternative to the function `search` use `sessionInfo` to see system packages and user attached packages.

```
> sessionInfo()
R version 2.5.0 (2007-04-23)
i386-pc-mingw32

locale:
LC_COLLATE=English_United States.1252;LC_CTYPE=English_United States.1252;
LC_MONETARY=English_United States.1252;LC_NUMERIC=C;
LC_TIME=English_United States.1252

attached base packages:
[1] "stats"     "graphics"  "grDevices" "datasets"  "tcltk"
[7] "utils"     "methods"   "base"

other attached packages:
    MASS svSocket     svIO   R2HTML   svMisc    svIDE
"7.2-33"  "0.9-5"  "0.9-5"   "1.58"  "0.9-5"  "0.9-5"
```

1.8 Conflicting objects

It is not recommended to do, but R allows the user to give an object a name that already exists. If you are not sure if a name already exists, just enter the name in the R console and see if R can find it. R will look for the object in all the libraries (packages) that are currently attached to the R system. R will not warn you when you use an existing name.

```
> mean = 10
> mean
[1] 10
```

The object `mean` already exists in the base package, but is now *masked* by your object `mean`. To get a list of all masked objects use the function `conflicts`.

```
> conflicts()
[1] "body<-" "mean"
```

You can safely remove the object `mean` with the function `rm` without risking deletion of the `mean` function. Calling `rm` removes only objects in your working environment by default.

1.9 Editors for R scripts

1.9.1 The editor in RGui

The console window in R is only useful when you want to enter one or two statements. It is not useful when you want to edit or write larger blocks of R code. In the RGui window you can open a new script, go to the 'File' menu and select 'New Script'. An empty R editor will appear where you can enter R code. This code can be saved, it will be a normal text file, normally with a .R extension. Existing text files with R code can be opened in the RGui window.

To run code in an R editor, select the code and use <Ctrl>-R to run the selected code. You can see that the code is parsed in the console window, any results will be displayed there.

1.9.2 Other editors

The built-in R editor is not the most fancy editor you can think of. It does not have much functionality. Since writing R code is just creating text files, you can do that with any text editor you like. If you have R code in a text file, you can use the `source` function to run the code in R. The function reads and executes all the statements in a text file.

```
# In the console window
source("C:\\Temp\\MyRfile.R")
```

There are free text editors that can send the R code inside the text editor to an R session. Some free editors that are worth mentioning are Eclipse (www.eclipse.org), Tinn-R (http://www.sciviews.org/Tinn-R) and JGR (speak 'Jaguar' http://jgr.markushelbig.org).

Eclipse

Eclipse is more than a text editor it is an environment to create, test manage and maintain (large) pieces of code. Built in functionality includes:

- Managing different text files in a project.
- Version control, recall previously saved versions of your text file.
- Search in multiple files.

The eclipse environment allows user to develop so called perspectives (or plug-ins). Such a plug-in customizes the Eclipse environment for a certain programming language. Stephan Wahlbrink has written an Eclipse plug-in for R, called 'StatEt'. See www.walware.de/goto/statet and see [1]. This plug-in adds extra 'R specific' functionality:

- Start an R console or terminal within Eclipse.
- Color coding of key words.
- Run R code in Eclipse by sending it to the R console.
- Insert predefined blocks of R code (templates).
- Supports writing R documentation files (*.Rd files).

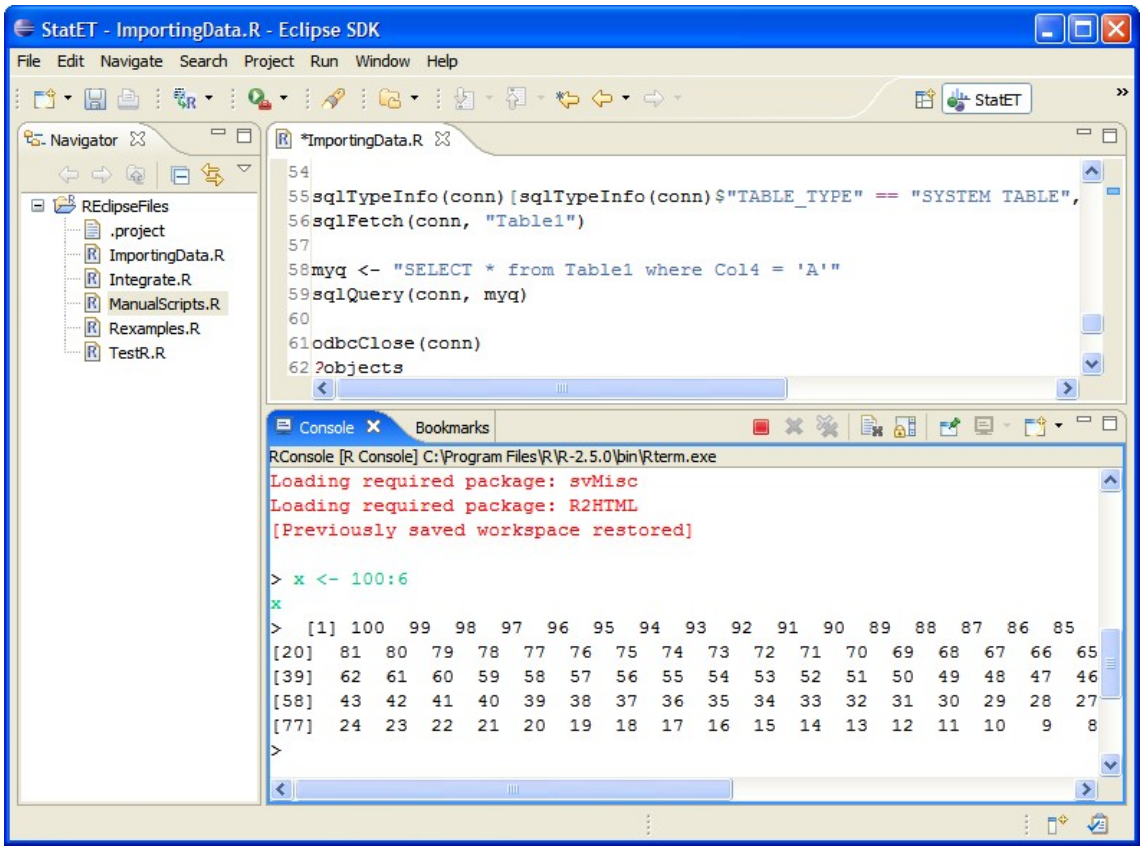

Figure 1.2: R integrated in the Eclipse development environment

Tinn-R

Tinn stands for Tinn is not Notepad, it is a text editor that was originally developed to replace the boring Notepad. With each new version of Tinn more features were added,

and it has become a very nice environment to edit and maintain code. Tinn-R is the special R version of Tinn. It allows color highlighting of the R language and sending R statements to an R Console window.

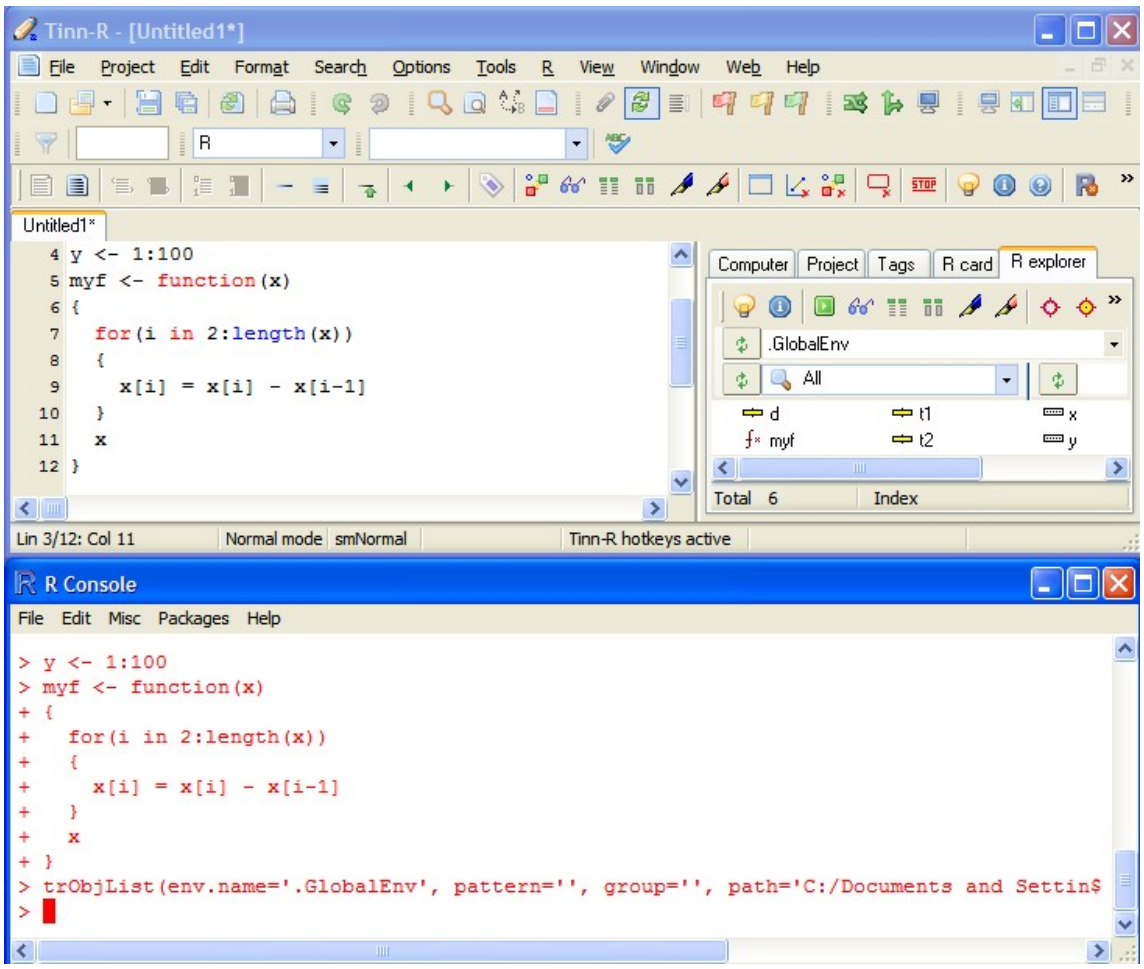

Figure 1.3: The Tinn-R and an the R Console environment

JGR

JGR (Java GUI for R) is a universal and unified Graphical User Interface for R. It includes among others: an integrated editor, help system 'Type-on' spreadsheet and an object browser.

2 Data Objects

In this section we will discuss the different aspects of data types and structures in R. Operators such as `c` and `:` will be used in this section as an illustration and will be discussed in the next section. If you are confronted with an unknown function, you can ask for help by typing in the command:

```
help(function name)
```

A help text will appear and describe the purpose of the function and how to use it.

2.1 Data types

2.1.1 Double

If you do calculations on numbers, you can use the data type double to represent the numbers. Doubles are numbers like `3.1415`, `8.0` and `8.1`. Doubles are used to represent continuous variables like the weight or length of a person.

```
x <- 8.14
y <- 8.0
z <- 87.0 + 12.9
```

Use the function `is.double` to check if an object is of type double. Alternatively, use the function `typeof` to ask R the type of the object `x`.

```
typeof(x)
[1] "double"
is.double(8.9)
[1] TRUE
test <- 1223.456
is.double(test)
[1] TRUE
```

Keep in mind that doubles are just approximations to real numbers. Mathematically there are infinity many numbers, the computer can ofcourse only represent a finite number of numbers. Not only can numbers like π or $\sqrt{2}$ not be represented exactly, less exotic numbers like 0.1 for example can also not be represented exactly.

One of the conseqeunces of this is that when you compare two doubles with each other you should take some care. Consider the following (surprising) result.

```
0.3 == 0.1 + 0.1 + 0.1
[1] FALSE
```

2.1.2 Integer

Integers are natural numbers. They can be used to represent counting variables, for example the number of children in a household.

```
nchild <- as.integer(3)
is.integer(nchild)
[1] TRUE
```

Note that 3.0 is not an integer, nor is 3 by default an integer!

```
nchild  <- 3.0
is.integer(nchild)
[1] FALSE
nchild <- 3
is.integer(nchild)
[1] FALSE
```

So a 3 of type 'integer' in R is something different than a 3.0 of type 'double'. However, you can mix objects of type 'double' and 'integer' in one calculation without any problems.

```
x <- as.integer(7)
y <- 2.0
z <- x/y
```

In contrast to some other programming languages, the answer is of type double and is 3.5. The maximum integer in R is $2^{31} - 1$.

```
as.integer(2^31 - 1)
[1] 2147483647
as.integer(2^31)
[1] NA
Warning message:
NAs introduced by coercion
```

2.1.3 Complex

Objects of type 'complex' are used to represent complex numbers. In statistical data analysis you will not need them often. Use the function `as.complex` or `complex` to create objects of type complex.

```
test1 <- as.complex(-25+5i)
sqrt(test1)
[1] 0.4975427+5.024694i

test2 <- complex(5,real=2,im=6)
test2
[1] 2+6i 2+6i 2+6i 2+6i 2+6i
typeof(test2)
[1] "complex"
```

Note that by default calculations are done on real numbers, so `sqrt(-1)` results in `NA`. Use

```
sqrt(as.complex(-1))
```

2.1.4 Logical

An object of data type logical can have the value `TRUE` or `FALSE` and is used to indicate if a condition is true or false. Such objects are usually the result of logical expressions.

```
x <- 9
y <- x > 10
y
[1] FALSE
```

The result of the function `is.double` is an object of type logical (TRUE or FALSE).

```
is.double(9.876)
[1] TRUE
```

Logical expressions are often built from logical operators:

`<`	smaller than
`<=`	smaller than or equal to
`>`	larger than
`>=`	larger than or equal to
`==`	is equal to
`!=`	is unequal to

The logical operators `and`, `or` and `not` are given by `&`, — and `!`, respectively.

```
x <- c(9,166)
y <- (3 < x) & (x <= 10)
[1] TRUE FALSE
```

Calculations can also be carried out on logical objects, in which case the `FALSE` is replaced by a zero and a one replaces the `TRUE`. For example, the sum function can be used to count the number of `TRUE`'s in a vector or array.

```
x <- 1:15
## number of elements in x larger than 9
sum(x>9)
[1] 6
```

2.1.5 Character

A character object is represented by a collection of characters between double quotes (`"`). For example: `"x"`, `"test character"` and `"iuiu8ygy-iuhu"`. One way to create character objects is as follows.

```
x <- c("a","b","c")
x
[1] "a" "b" "c"
mychar1 <- "This is a test"
mychar2 <- "This is another test"
charvector <- c("a", "b", "c", "test")
```

The double quotes indicate that we are dealing with an object of type 'character'.

2.1.6 Factor

The factor data type is used to represent categorical data (i.e. data of which the value range is a collection of codes). For example:

- variable 'sex' with values male and female.
- variable 'blood type' with values: A, AB and O.

An individual code of the value range is also called a *level* of the factor variable. So the variable 'sex' is a factor variable with two levels, male and female.

Sometimes people confuse factor type with character type. Characters are often used for labels in graphs, column names or row names. Factors must be used when you want to represent a discrete variable in a data frame and want to analyze it.

Factor objects can be created from character objects or from numeric objects, using the function factor. For example, to create a vector of length five of type factor do the following:

```
sex <- c("male","male","female","male","female")
```

The object `sex` is a character object. You need to transform it to factor.

```
sex <- factor(sex)
sex
[1] male   male   female  male   female
```

Use the function `levels` to see the different levels a factor variable has.

```
levels(sex)
[1] "female" "male"
```

Note that the result of the `levels` function is of type character. Another way to generate the sex variable is as follows:

```
sex <- c(1,1,2,1,2)
```

The object 'sex' is an integer variable, you need to transform it to a factor.

```
sex <- factor(sex)
sex
[1] 1 1 2 1 2
Levels: 1 2
```

The object 'sex' looks like, but is not an integer variable. The 1 represents level "1" here. So arithmetic operations on the sex variable are not possible:

```
sex + 7
[1] NA NA NA NA NA
Warning message:
+ not meaningful for factors in: Ops.factor(sex, 7)
```

It is better to rename the levels, so level "1" becomes male and level "2" becomes female:

```
levels(sex) <- c("male","female")
sex
[1] male   male   female male   female
```

You can transform factor variables to double or integer variables using the `as.double` or `as.integer` function.

```
sex.numeric <- as.double(sex)
sex.numeric
[1] 2 2 1 2 1
```

The 1 is assigned to the female level, only because alphabetically female comes first. If the order of the levels is of importance, you will need to use *ordered factors*. Use the function `ordered` and specify the order with the levels argument. For example:

```
Income <- c("High","Low","Average","Low","Average","High","Low")
Income <- ordered(Income, levels=c("Low","Average","High"))
Income
[1] High      Low      Average Low   Average High    Low
Levels: Low < Average < High
```

The last line indicates the ordering of the levels within the factor variable. When you transform an ordered factor variable, the order is used to assign numbers to the levels.

```
Income.numeric  <- as.double(Income)
Income.numeric
[1] 3 1 2 1 2 3 1
```

The order of the levels is also used in linear models. If one or more of the regression variables are factor variables, the order of the levels is important for the interpretation of the parameter estimates see section 8.3.4.

2.1.7 Dates and Times

To represent a calendar date in R use the function `as.Date` to create an object of class Date.

```
temp <- c("12-09-1973", "29-08-1974")
z <- as.Date(temp, "%d-%m-%Y")
z
[1] "1973-09-12" "1974-08-29"
data.class(z)
[1] "Date"
format(z, "%d-%m-%Y")
[1] "12-09-1973" "29-08-1974"
```

You can add a number to a date object, the number is interpreted as the number of day to add to the date.

```
z + 19
[1] "1973-10-01" "1974-09-17"
```

You can subtract one date from another, the result is an object of class 'difftime'

```
dz = z[2] -z[1]
dz
data.class(dz)
Time difference of 351 days
[1] "difftime"
```

In R the classes `POSIXct` and `POSIXlt` can be used to represent calendar dates and times. You can create `POSIXct` objects with the function `as.POSIXct`. The function accepts characters as input, and it can be used to not only to specify a date but also a time within a date.

```
t1 <- as.POSIXct("2003-01-23")
t2 <- as.POSIXct("2003-04-23 15:34")
t1
t2
[1] "2003-01-23 W. Europe Standard Time"
[1] "2003-04-23 15:34:00 W. Europe Daylight Time"
```

A handy function is `strptime`, it is used to convert a certain character representation of a date (and time) into another character representation. You need to provide a conversion specification that starts with a % followed by a single letter.

```
# first creating four characters
x <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
z <- strptime(x, "%d%b%Y")
zt <- as.POSIXct(z)
zt
[1] "1960-01-01 W. Europe Standard Time"
[2] "1960-01-02 W. Europe Standard Time"
[3] "1960-03-31 W. Europe Daylight Time"
[4] "1960-07-30 W. Europe Daylight Time"

# pasting 4 character dates and 4 character times together
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92")
times <- c("23:03:20", "22:29:56", "01:03:30", "18:21:03")
x <- paste(dates, times)
z <- strptime(x, "%m/%d/%y %H:%M:%S")
zt <- as.POSIXct(z)
zt
[1] "1992-02-27 23:03:20 W. Europe Standard Time"
[2] "1992-02-27 22:29:56 W. Europe Standard Time"
[3] "1992-01-14 01:03:30 W. Europe Standard Time"
[4] "1992-02-28 18:21:03 W. Europe Standard Time"
```

An object of type `POSIXct` can be used in certain calculations, a number can be added to a `POSIXct` object. This number will be the interpreted as the number of seconds to add to the POSIXct object.

```
zt + 13
[1] "1992-02-27 23:03:33 W. Europe Standard Time"
[2] "1992-02-27 22:30:09 W. Europe Standard Time"
[3] "1992-01-14 01:03:43 W. Europe Standard Time"
[4] "1992-02-28 18:21:16 W. Europe Standard Time"
```

You can subtract two POSIXct objects, the result is a so called 'difftime' object.

```
t2 <- as.POSIXct("2004-01-23 14:33")
t1 <- as.POSIXct("2003-04-23")
d <- t2-t1
d
Time difference of 275.6479 days
```

A 'difftime' object can also be created using the function `as.difftime`, and you can add a difftime object to a POSIXct object. Due to a bug in R this can only safely be done with the function `"+.POSIXt"`.

```
"+.POSIXt"(zt, d)
[1] "1992-11-29 14:36:20 W. Europe Standard Time"
[2] "1992-11-29 14:02:56 W. Europe Standard Time"
[3] "1992-10-15 17:36:30 W. Europe Daylight Time"
[4] "1992-11-30 09:54:03 W. Europe Standard Time"
```

To extract the weekday, month or quarter from a POSIXct object use the handy R functions `weekdays`, `months` and `quarters`. Another handy function is `Sys.time`, which returns the current date and time.

```
weekdays(zt)
[1] "Thursday" "Thursday" "Tuesday"  "Friday"
```

There are some R packages that can handle dates and time objects. For example, the packages zoo, chron, tseries, its and Rmetrics. Especially Rmetrics has a set of powerful functions to maintain and manipulate dates and times. See [2].

2.1.8 Missing data and Infinite values

We have already seen the symbol `NA`. In R it is used to represent 'missing' data (*Not Available*). It is not really a separate data type, it could be a missing double or a missing integer. To check if data is missing, use the function `is.na` or use a direct comparison with the symbol NA. There is also the symbol `NaN` (*Not a Number*), which can be detected with the function `is.nan`.

```
x <- as.double( c("1", "2", "qaz"))
is.na(x)
[1] FALSE FALSE  TRUE

z <- sqrt(c(1,-1))
Warning message:
NaNs produced in: sqrt(c(1, -1))
is.nan(z)
[1] FALSE  TRUE
```

Infinite values are represented by `Inf` or `-Inf`. You can check if a value is infinite with the function `is.infinite`. Use `is.finite` to check if a value is finite.

```
x <- c(1,3,4)
y <- c(1,0,4)
x/y
[1]   1 Inf   1
```

```
z <- log(c(4,0,8))
is.infinite(z)
[1] FALSE  TRUE FALSE
```

In R `NULL` represents the null object. `NULL` is used mainly to represent the lists with zero length, and is often returned by expressions and functions whose value is undefined.

2.2 Data structures

Before you can perform statistical analysis in R, your data has to be structured in some coherent way. To store your data R has the following structures:

- vector
- matrix
- array
- data frame
- time-series
- list

2.2.1 Vectors

The simplest structure in R is the vector. A vector is an object that consists of a number of elements of the same type, all doubles or all logical. A vector with the name 'x' consisting of four elements of type 'double' (10, 5, 3, 6) can be constructed using the function `c`.

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

The function `c` merges an arbitrary number of vectors to one vector. A single number is regarded as a vector of length one.

```
y <- c(x,0.55, x, x)
y
[1] 10.0 5.0 3.0 6.0 0.55 10.0 5.0 3.0 6.0
[10] 10.0 5.0 3.0 6.0
```

Typing the name of an object in the commands window results in printing the object. The numbers between square brackets indicate the position of the following element in the vector.

Use the function `round` to round the numbers in a vector.

```
round (y,3)  # round to 3 decimals
```

Mathematical calculations on vectors

Calculations on (numerical) vectors are usually performed on each element. For example, `x*x` results in a vector which contains the squared elements of `x`.

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

The symbols for elementary arithmetic operations are `+`, `-`, `*`, `/`. Use the `^` symbol to raise power. Most of the standard mathematical functions are available in R. These functions also work on each element of a vector. For example the logarithm of x:

```
log(x)
[1] 2.302585 1.609438 1.098612 1.791759
```

Function name	Operation
`abs`	absolute value
`asin acos atan`	inverse geometric functions
`asinh acosh atanh`	inverse hyperbolic functions
`exp log`	exponent and natural logarithm
`floor ceiling trunc`	creates integers from floating point numbers
`gamma lgamma`	gamma and log gamma function
`log10`	logarithm with basis 10
`round`	rounding
`sin cos tan`	geometric functions
`sinh cosh tanh`	hyperbolic functions
`sqrt`	square root

Table 2.1: Some mathematical functions that can be applied on vectors

The recycling rule

It is not necessary to have vectors of the same length in an expression. If two vectors in an expression are not of the same length then the shorter one will be repeated until it has the same length as the longer one. A simple example is a vector and a number (which is a vector of length one).

```
sqrt(x) + 2
[1] 5.162278 4.236068 3.732051 4.449490
```

In the above example the 2 is repeated 4 times until it has the same length as `x` and then the addition of the two vectors is carried out. In the next example, `x` has to be repeated 1.5 times in order to have the same length as `y`. This means the first two elements of `x` are added to `x` and then `x*y` is calculated.

```
x <- c(1,2,3,4)
y <- c(1,2,3,4,5,6)
z <- x*y
Warning message:
longer object length
        is not a multiple of shorter object length in: x * y
> z
[1]  1  4  9 16  5 12
```

Generating vectors

Regular sequences of numbers can be very handy for all sorts of reasons. Such sequences can be generated in different ways. The easiest way is to use the column operator (`:`).

```
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
```

A descending sequence is obtained by `20:1`. The function seq together with its arguments from, to, by or length is used to generate more general sequences. Specify the beginning and end of the sequence and either specify the length of the sequence or the increment.

```
u <- seq(from=-3,to=3,by =0.5)
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
```

The following commands have the same result:

```
u <- seq(-3,3,length=13)
u <- (-6):6/2
```

The function `seq` can also be used to generate vectors with POSIXct elements (a sequence of dates). The following examples speak for them selves.

```
seq(as.POSIXct("2003-04-23"), by = "month", length = 12)
[1] "2003-04-23 W. Europe Daylight Time" "2003-05-23 W. Europe Daylight Time"
[3] "2003-06-23 W. Europe Daylight Time" "2003-07-23 W. Europe Daylight Time"
...
seq(ISOdate(1910,1,1), ISOdate(1999,1,1), "years")
[1] "1910-01-01 12:00:00 GMT" "1911-01-01 12:00:00 GMT"
[3] "1912-01-01 12:00:00 GMT" "1913-01-01 12:00:00 GMT"
...
```

The function `rep` repeats a given vector. The first argument is the vector and the second argument can be a number that indicates how often the vector needs to be 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 can also be a vector of the same length as the vector used for the first argument. In this case each element in the second vector indicates how often the corresponding element in the first vector is repeated.

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

For information about other options of the function `rep` type `help(rep)`. To generate vectors with random elements you can use the functions `rnorm` or `runif`. There are more of these functions.

```
x <- rnorm(10) # 10 random standard normal numbers
y <- runif(10,4,7) # 10 random numbers between 4 and 7
```

2.2.2 Matrices

Generating matrices

A matrix can be regarded as a generalization of a vector. As with vectors, all the elements of a matrix must be of the same data type. A matrix can be generated in several ways. For example:

- Use the function `dim`:

```
x <- 1:8
dim(x) <- c(2,4)
x
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

- Use the function `matrix`:

```
x <- matrix(1:8,2,4,byrow=F)
x
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

By default the matrix is filled by column. To fill the matrix by row specify `byrow = T` as argument in the matrix function.

1. Use the function `cbind` to create a matrix by binding two or more vectors as column vectors. The function `rbind` is used to create a matrix by binding two or more vectors as row vectors.

```
cbind(c(1,2,3),c(4,5,6))
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

rbind(c(1,2,3),c(4,5,6))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Calculations on matrices

A matrix can be regarded as a number of equal length vectors pasted together. All the mathematical functions that apply to vectors also apply to matrices and are applied on each matrix element.

```
x*x^2   # All operations are applied on each matrix element
```

```
     [,1] [,2] [,3] [,4]
[1,]    1   27  125  343
[2,]    8   64  216  512
```

```
max(x)  # returns the maximum of all matrix elements in x
[1]  8
```

You can multiply a matrix with a vector. The outcome may be surprising:

```
x <- matrix(1:16,ncol=4)
y <- 7:10
x*y
```

```
     [,1] [,2] [,3] [,4]
[1,]    7   35   63   91
[2,]   16   48   80  112
[3,]   27   63   99  135
[4,]   40   80  120  160
```

```
x <- matrix(1:28,ncol=4)
y <- 7:10
x*y
```

```
     [,1] [,2] [,3] [,4]
[1,]    7   80  135  176
[2,]   16   63  160  207
[3,]   27   80  119  240
[4,]   40   99  144  175
[5,]   35  120  171  208
[6,]   48   91  200  243
[7,]   63  112  147  280
```

As an exercise: try to find out what R did.

To perform a matrix multiplication in the mathematical sense, use the operator: `%*%`. The dimensions of the two matrices must agree. In the following example the dimensions are wrong:

```
x <- matrix(1:8,ncol=2)
x %*% x
Error in x %*% x : non-conformable arguments
```

A matrix multiplied with its transposed `t(x)` always works.

```
x %*% t(x)
[,1] [,2] [,3] [,4]
[1,]   26   32   38   44
[2,]   32   40   48   56
[3,]   38   48   58   68
[4,]   44   56   68   80
```

R has a number of matrix specific operations, for example:

Function name	Operation
`chol(x)`	Choleski decomposition
`col(x)`	matrix with column numbers of the elements
`diag(x)`	create a diagonal matrix from a vector
`ncol(x)`	returns the number of columns of a matrix
`nrow(x)`	returns the number of rows of a matrix
`qr(x)`	QR matrix decomposition
`row(x)`	matrix with row numbers of the elements
`solve(A,b)`	solve the system Ax=b
`solve(x)`	calculate the inverse
`svd(x)`	singular value decomposition
`var(x)`	covariance matrix of the columns

Table 2.2: Some functions that can be applied on matrices

A detailed description of these functions can be found in the corresponding help files, which can be accessed by typing for example `?diag` in the R Console.

2.2.3 Arrays

Arrays are generalizations of vectors and matrices. A vector is a one-dimensional array and a matrix is a two dimensional array. As with vectors and matrices, all the elements of an array must be of the same data type. An example of an array is the three-dimensional array 'iris3', which is a built-in data object in R. A three dimensional array can be regarded as a block of numbers.

```
dim(iris3)   # dimensions of iris
[1] 50  4  3
```

All basic arithmetic operations which apply to matrices are also applicable to arrays and are performed on each element.

```
test <- iris + 2*iris
```

The function `array` is used to create an array object

```
newarray <- array(c(1:8, 11:18, 111:118), dim = c(2,4,3))
newarray
```

```
, , 1
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8

, , 2
     [,1] [,2] [,3] [,4]
[1,]   11   13   15   17
[2,]   12   14   16   18

, , 3
     [,1] [,2] [,3] [,4]
[1,]  111  113  115  117
[2,]  112  114  116  118
```

2.2.4 Data frames

Data frames can also be regarded as an extension to matrices. Data frames can have columns of different data types and are the most convenient data structure for data analysis in R. In fact, most statistical modeling routines in R require a data frame as input.

One of the built-in data frames in R is 'mtcars'.

```
mtcars

  # only a small part of mtcars
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
```

The data frame contains information on different cars. Usually each row corresponds with a case and each column represents a variable. In this example the 'carb' column is of data type 'double' and represents the number of carburetors. See the help file for more information on this data frame; `?mtcars`.

Data frame attributes

A data frame can have the attributes names and row.names. The attribute names contains the column names of the data frame and the attribute row.names contains the row names of the data frame. The attributes of a data frame can be retrieved separately from the data frame with the functions `names` and `row.names`. The result is a character vector containing the names.

```
rownames(mtcars)[1:5] # only the first five row names
[1] "Mazda RX4"         "Mazda RX4 Wag"     "Datsun 710"
[4] "Hornet 4 Drive"    "Hornet Sportabout"

 names(mtcars)
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
```

Creating data frames

You can create data frames in several ways, by importing a data file as in Chapter 3, for example, or by using the function `data.frame`. This function can be used to create new data frames or convert other objects into data frames.

A few examples of the `data.frame` function:

```
my.logical <- sample(c(T,F),size=20,replace = T)
my.numeric <- rnorm(20)
my.df <- data.frame(my.logical,my.numeric)
my.df

   my.logical  my.numeric
 1       TRUE  0.63892503
 2       TRUE -1.14575124
 3       TRUE -1.27484164
..     ..       ..
..     ..       ..
19       TRUE -0.01115154
20       TRUE -1.07818944
```

```
test <- matrix(rnorm(21),7,3) # create a matrix with random elements
test <- data.frame(test)
test
       X1         X2     X3
1 -0.36428978  0.63182432  0.6977597
2 -0.24943864 -1.05139082 -0.9063837
3  0.95472560 -0.46806163  1.0057703
4  0.48152529 -2.03857066 -0.7163017
5 -0.71593428 -2.18493234 -2.7043682
6 -1.20729385 -0.50772018  1.1240321
7 -0.07551876  0.06711515  0.1897599

names(test)
[1] "X1" "X2" "X3"
```

R automatically creates column names: 'X1', 'X2' and 'X3'. You can use the `names` function to change these column names.

```
names(test) <- c("Price", "Length", "Income")

row.names(test) <- c("Paul","Ian","Richard","David","Rob","Andrea","John")
test
             Price      Length   Income
   Paul -0.36428978  0.63182432  0.6977597
    Ian -0.24943864 -1.05139082 -0.9063837
Richard  0.95472560 -0.46806163  1.0057703
  David  0.48152529 -2.03857066 -0.7163017
    Rob -0.71593428 -2.18493234 -2.7043682
 Andrea -1.20729385 -0.50772018  1.1240321
   John -0.07523445  0.32454334  1.3432442
```

2.2.5 Time-series objects

In R a time-series object (an object of class 'ts') is created with the function `ts`. It combines two components:

- The data, a vector or matrix of numeric values. In case of a matrix, each column is a separate time-series.
- The dates of the data, the dates are equispaced points in time.

```
# starting from jan-87, 100 monthly intervals
myts1 <- ts(data = rnorm(100), start=c(1987), freq = 12)
# two time-series starting from apr-1987, 50 monthly intervals
```

```
myts2 <- ts(data = matrix(rnorm(100),ncol=2), start=c(1987,4), freq=12)
myts2
            Series 1   Series 2
Apr 1987  1.66394678  1.3009008
May 1987 -0.48923748 -0.8199132
Jun 1987  0.21643666 -0.1581245
Jul 1987 -2.21148119 -0.4926389
Aug 1987  0.26117051  1.1255435
...
```

The function `tsp` returns the start and end time, and also the frequency without printing the complete data of the time-series.

```
tsp(myts2)
[1] 1987.250 1991.333   12.000
```

2.2.6 Lists

A list is like a vector. However, an element of a list can be an object of any type and structure. Consequently, a list can contain another list and therefore it can be used to construct arbitrary data structures. Lists are often used for output of statistical routines in R. The output object is often a collection of parameter estimates, residuals, predicted values etc.

For example, consider the output of the function `lsfit`. In its most simple form the function fits a least square regression.

```
x <- 1:5
y <- x + rnorm(5,0,0.25)
z <- lsfit(x,y)
z

$coef:
 Intercept         X
 0.1391512 0.9235291

$residuals:
[1] -0.006962623 -0.017924751 -0.036747141  0.155119026 -0.093484512

$intercept:
[1] T
```

In this example the output value of `lsfit(x,y)` is assigned to object 'z'. This is a list whose first component is a vector with the intercept and the slope. The second component is a vector with the model residuals and the third component is a logical vector of length one indicating whether or not an intercept is used. The three components have the names 'coef', 'residuals' and 'intercept'.

The components of a list can be extracted in several ways:

- component number: `z[[1]]` means the first component of `z` (use double square brackets!).
- component name: `z$name` indicates the component of `z` with name `name`.

To identify the component name the first few characters will do, for example, you can use `z$r` instead of `z$residuals`.

```
test <- z$r
test
[1] -0.0069626 -0.0179247 -0.0367471  0.1551190 -0.0934845
z$r[4]  # fourth element of the residuals
[1] 0.155119026
```

Creating lists

A list can also be constructed by using the function `list`. The names of the list components and the contents of list components can be specified as arguments of the `list` function by using the = character.

```
x1 <- 1:5
x2 <- c(T,T,F,F,T)
y <- list(numbers=x1, wrong=x2)
y
$numbers
[1] 1 2 3 4 5

$wrong
[1]  TRUE  TRUE FALSE FALSE  TRUE
```

So the left-hand side of the = operator in the `list` function is the name of the component and the right-hand side is an R object. The order of the arguments in the list function determines the order in the list that is created. In the above example the logical object 'wrong' is the second component of `y`.

```
y[[2]]
[1]  TRUE  TRUE FALSE FALSE  TRUE
```

The function `names` can be used to extract the names of the list components. It is also used to change the names of list components.

```
names(y)
[1] "numbers"   "wrong"
names(y) <- c("lots", "valid")
names(y)
[1] "lots"  "valid"
```

To add extra components to a list proceed as follows:

```
y[[3]] <- 1:50

y$test <- "hello"

y
$lots
[1] 1 2 3 4 5

$valid
[1]  TRUE  TRUE FALSE FALSE  TRUE

[[3]]
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50

$test:
[1] "hello"
```

Note the difference in single square brackets and double square brackets.

```
y[1]
$numbers:
[1] 1 2 3 4 5

y[[1]]
[1] 1 2 3 4 5
```

When single square brackets are used, the component is returned as list, whereas double square brackets return the component itself.

Transforming objects to a list

Many objects can be transformed to a list with the function `as.list`. For example, vectors, matrices and data frames.

```
as.list(1:6)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3
...
...
```

2.2.7 The `str` function

A handy function is the `str` function, it displays the internal structure of an R object. The function can be used to see a short summary of an object.

```
x1 <- rnorm(1000)
x2 <- matrix(rnorm(80000),ncol=80)
myl1 <- list(x1,x2,my.df)

str(x1)
num [1:1000]  2.326  1.889  1.740 -1.008  0.916 ...
str(x2)
num [1:1000, 1:80] -0.0368 -0.2626  0.8323 -0.3204  0.2559 ...
str(my.df)
'data.frame': 20 obs. of  2 variables:
 $ my.logical: logi   TRUE  TRUE  TRUE  TRUE  TRUE FALSE ...
 $ my.numeric: num   0.0079 -0.0480  0.4988  0.2047 -0.6340 ...
str(myl1)
List of 3
 $ : num [1:1000]  2.326  1.889  1.740 -1.008  0.916 ...
 $ : num [1:1000, 1:80] -0.0368 -0.2626  0.8323 -0.3204  0.2559 ...
 $ :'data.frame': 20 obs. of  2 variables:
  ..$ my.logical: logi [1:20]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE ...
  ..$ my.numeric: num [1:20]  0.0079 -0.0480  0.4988  0.2047 -0.6340 ...
```

3 Importing data

One of the first things you want to do in a statistical data analysis system is to import data. R provides a few methods to import data, we will discuss them in this chapter.

3.1 Text files

In R you can import text files with the function `read.table`. This function is a has many arguments. Arguments to specify the header, the column separator, the number of lines to skip, the data types of the columns, etc. The functions `read.csv` and `read.delim` are functions to read 'comma separated values' files and tab delimited files. These functions call `read.table` with specific arguments.

Suppose we have a text file data.txt, that contains the following text:

```
Author: John Davis
Date: 18-05-2007
Some comments..
Col1, Col2, Col3, Col4
23, 45, A, John
34, 41, B, Jimmy
12, 99, B, Patrick
```

The data without the first few lines of text can be imported to an R data frame using the following R syntax:

```
myfile <- "C:\\Temp\\R\\Data.txt"
mydf <- read.table(myfile, skip=3, sep=",", header=TRUE)
mydf
   Col1 Col2 Col3    Col4
1   23   45    A     John
2   34   41    B    Jimmy
3   12   99    B  Patrick
```

By default R converts character data in text files to factor type. In the above example the third and fourth columns are of type factor. To leave character data as character data type in R, use the `stringsAsFactors` argument.

```
mydf <- read.table(
          myfile,
          skip=3,
          sep=",",
          header=TRUE,
          stringsAsFactors=FALSE
        )
```

To specify that certain columns are character and other columns are not you must use the `colClasses` argument and provide the type for each column.

```
mydf <- read.table(
          myfile, skip=3, sep=",",
          header=TRUE, stringsAsFactors=FALSE,
          colClasses = c("numeric", "numeric", "factor", "character")
        )
```

There is an advantage in using `colClasses`, especially when the data set is large. If you don't use `colClasses` then during a data import, R will store the data as character vectors before deciding what to do with them.

Character strings in a text files may be quoted and may contain the the separator symbol. To import such text files use the `quote` argument. Suppose we have the following comma separated text file that we want to read.

```
Col1, Col2, Col3
12, 45, 'Davis, Joe'
23, 78, 'White, Jimmy'
```

Use the `read.csv` function as follows to import the above text.

```
read.csv(myfile, quote="'")
  Col1 Col2          Col3
1   12   45    Davis, Joe
2   23   78  White, Jimmy
```

3.1.1 The `scan` function

The `read.table` function uses the more low level function `scan`. This function may also be called directly by the user, and can sometimes be handy when `read.table` cannot do the job. It reads the data into a vector or list, the user can then manipulate this vector or list. For example, if we use `scan` to read the text file above we get:

```
scan(myfile, what="character", sep=",", strip.white =TRUE)
[1] "Col1"         "122"          "Col3"         "12"           "45"
[6] "Davis, Joe"   "23"           "78"           "White, Jimmy"
Read 9 items
```

3.2 Excel files

To read and write Excel files you can use the package `xlsReadWrite`. This package provides the functions `read.xls` and `write.xls`. If the data is in the first sheet and starts at row 1, where the first row represent the column headers, then the call to `read.xls` is simple.

```
library("xlsReadWrite")
myfile <- "C:\\RFiles\\ExcelData.xls"
mydf <- read.xls(myfile)
mydf
  Col1 Col2  Col3   Col4
1   12    A 26919   john
2   23    A 33077 martin
3    5    B 31788   adam
4   56    C 30176  clair
```

The function `read.xls` uses the R default to determine if strings (characters) in the Excel data should be converted to factors. There are two ways to import strings as character in R.

```
# all string data is converted to character type
mydf <- read.xls(myfile, stringsAsFactors = T)
# specify the type of each column
mydf <- read.xls(myfile,
           colClasses = c(
              "numeric",
              "factor",
              "isodatetime",
```

```
            "character"
        )
    )
```

Use the arguments `sheet` and `from` to import data from different works sheets and starting rows.

3.3 Databases

There are several R packages that support the import (and export) of data from databases.

- Package `RODBC`, provides an interface to databases that are ODBC compliant. These include, MS SQLServer, MS Access, Oracle.
- Package `RMySQL`, provides an interface to the MySQL database
- Package `RJDBC`, provides an interface to databases that are JDBC compliant.
- Package `RSQLite`, not only interfaces this package with SQLite, it embeds the SQLite engine in R.

We give a small example to import a table in R from an MS-Access database using ODBC. An important step is to set up 'Data Source Name' (DSN) using the administrative tools in Windows. Once that is done, R can import data from the corresponding database.

- Go to the 'Control Panel', select 'Administrative Tools' and select 'Data Sources (ODBC)'.
- In the tab 'User DSN' click the 'Add' button, select the MS Access driver and click 'Finish'
- Now chose a name for the data source, say, 'MyAccessData' and select the MS Access database file.

Now the DSN has been set up and we can import the data from the database into R. First make a connection object using the function `odbcConnect`.

```
library(RODBC)
conn <- odbcConnect("MyAccessData")
conn
RODB Connection 1
Details:
  case=nochange
  DSN=MyAccessData
  DBQ=C:\DOCUMENTS AND SETTINGS\LONGHOW LAM\My
```

```
   Documents\LonghowStuff\Courses\R\MyAccess.mdb
  DriverId=25
  FIL=MS Access
  MaxBufferSize=2048
  PageTimeout=5
```

If you have established a connection successfully, the connection object will display a summary of the connection. To display table information use `sqlTables(conn)`, which will display all tables, including system tables. To import a specific table use the function `sqlFetch`.

```
sqlFetch(conn, "Table1")
  ID   Col1 Col2       Col3 Col4
1  1   John  123 1973-09-12    A
2  2 Martin  456 1999-12-31    B
3  3  Clair  345 1978-05-22    B
```

Use the function `sqlQuery` to submit an SQL query to the database and retrieve the result.

```
myq <- "SELECT * from Table1 where Col4 = 'A'"
sqlQuery(conn, myq)
  ID Col1 Col2       Col3 Col4
1  1 John  123 1973-09-12    A
```

You can have multiple connections to multiple databases, that can be useful if you need to collect and merge data from several sources. The function `odbcDataSources` lists all the available data sources. Don't forget to close a connection with `odbcClose(conn)`.

3.4 The Foreign package

4 Data Manipulation

The programming language in R provides many different functions and mechanisms to manipulate and extract data. Let's look at some of those for the different data structures.

4.1 Vector subscripts

A part of a vector `x` can be selected by a general subscripting mechanism.

```
x[subscript]
```

The simplest example is to select one particular element of a vector, for example the first one or the last one.

```
x <- c(6,7,2,4)
x[1]
[1] 6
x[length(x)]
[1] 4
```

Moreover, the subscript can have one of the following forms:

A vector of positive natural numbers The elements of the resulting vector are determined by the numbers in the subscript. To extract the first three numbers:

```
x
[1] 10 5 3 6
x[1:3]
[1] 10 5 3
```

To get a vector with the fourth, first and again fourth element of `x`:

```
x[c(4,1,4)]
[1] 6 10 6
```

One or more elements of a vector can be changed by the subscripting mechanism. To change the third element of a vector proceed as follows:

```
x[3] <- 4
```

To change the first three elements:

```
x[1:3] <- 4
```

The last two constructions are examples of a so-called replacement, in which the left hand side of the assignment operator is more than a simple identifier. Note also that the recycling rule applies, so the following code works (with a warning from R).

```
x[1:3] <- c(1,2)
```

A logical vector The result is a vector with only those elements of `x` of which the logical vector has an element `TRUE`.

```
x <- c(10,4,6,7,8)
y <- x >9
y
[1]  TRUE FALSE FALSE FALSE FALSE
x[y]
[1] 10
```

or directly

```
x[x>9]
[1] 10
```

To change the elements of `x` which are larger than 9 to the value 9 do the following:

```
x[x>9] <- 9
```

Note that the logical vector does not have to be of the same length as the vector you want to extract elements from.

A vector of negative natural numbers All elements of `x` are selected except those that are in the subscript.

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

Note the subscript vector may address non-existing elements of the original vector. The result will be `NA` (Not Available). For example,

```
x <- c(1,2,3,4,5)
x[7]
[1] NA

x[1:6]
[1] 1 2 3 4 5 NA
```

Some useful functions There are several useful R functions for working with vectors.

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

These functions are used to calculate the length, the sum, the product, the minimum and the maximum of a vector, respectively. The last four functions can also be used on more than one vector, in which case the sum, product, minimum, or maximum is taken over all elements of all vectors.

```
x <- 10:71
y <- 45:21
sum(x,y); prod(x,y); max(x,y); min(x,y)
## chop off last part of a vector
x <- 10:100
length(x) = 20
```

Note that `sum(x,y)` is equal to `sum(c(x,y))`.

The function `cumsum(x)` generates a vector with the same length as the input vector. The i-th element of the resulting vector is equal to the sum of the first i elements of the input vector. Example:

```
cumsum(rep(2,10))
 [1]  2  4  6  8 10 12 14 16 18 20
```

To sort a vector in increasing order, use the function `sort`. You can also use this function to sort in decreasing order by using the argument `decrease = TRUE`.

```
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
sort(x, decr = TRUE)
 [1] 8 8 6 5 5 4 3 2 1 0
```

With the function `order` you can produce a permutation vector which indicates how to sort the input vector in ascending order. If you have two vectors `x` and `y`, you can sort `x` and permute `y` in such a way that the elements have the same order as the sorted vector `x`.

```
x <- rnorm(10) # create 10 random numbers
y <- 1:10      # create the numbers 1,2,3,..,10
z <- order(x)  # create a permutation vector
sort(x)        # sort x

[1] -1.069 -0.603 -0.554  0.872  0.942  0.972  1.083  1.924  2.194  2.456

y[z]  # change the order of elements of y

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

Try to figure out what the result of `x[order(x)]` is!

The function `rev` reverses the order of vector elements. So `rev(sort(x))` is a sorted vector in descending order.

```
x <- rnorm(10)
round( rev(sort(x)),2)
[1]  1.18  1.00  0.87  0.57 -0.37 -0.42 -0.49 -0.72 -0.91 -1.26
```

The function `unique` returns a vector which only contains the unique values of the input vector. The function `duplicated` returns for every element a `TRUE` or `FALSE` depending on whether or not that element has previously appeared in the vector.

```
x <- c(2,6,4,5,5,8,8,1,3,0)
unique(x)
[1] 2 6 4 5 8 1 3 0

duplicated(x)
[1] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
```

Our last example of a vector manipulation function is the function `diff`. It returns a vector which contains the differences between the consecutive input elements.

```
x <- c(1,3,5,8,15)
diff(x)
[1] 2 2 3 7
```

So the resulting vector of the function `diff` is always at least one element shorter than the input vector. An additional `lag` argument can be used to specify the lag of differences to be calculated.

```
x <- c(1,3,5,8,15)
diff(x, lag=2)
[1]  4  5 10
```

So in this case with `lag=2`, the resulting vector is two elements shorter.

4.2 Matrix subscripts

As with vectors, parts of matrices can be selected by the subscript mechanism. The general scheme for a matrix `x` is given by:

```
x[subscript]
```

Where subscript has one of the following four forms:

1. A pair `(rows, cols)` where `rows` is a vector representing the row numbers and `cols` is a vector representing column numbers. Rows and/or cols can be empty or negative. The following examples will illustrate the different possibilities.

```
x <- matrix(1:36, ncol=6)
## the element in row 2 and column 6 of x
x[2,6]
[1] 32

## the third row of x
x[3, ]
[1]  3  9 15 21 27 33

## the element in row 3 and column 1 and
## the element in row 3 and column 5
```

```
x[3,c(1,5)]
[1] 3 27

## show x, except for the first column
x[,-1]

      [,1] [,2] [,3] [,4] [,5]
[1,]    7   13   19   25   31
[2,]    8   14   20   26   32
[3,]    9   15   21   27   33
[4,]   10   16   22   28   34
[5,]   11   17   23   29   35
[6,]   12   18   24   30   36
```

A negative pair results in a so-called minor matrix where a column and row is omitted.

```
x[-3,-4]
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   25   31
[2,]    2    8   14   26   32
[3,]    4   10   16   28   34
[4,]    5   11   17   29   35
[5,]    6   12   18   30   36
```

The matrix `x` remains the same, unless you assign the result back to `x`.

```
x <- x[-3,4]
```

As with vectors, matrix elements or parts of matrices can be changed by using the matrix subscript mechanism and the assignment operator together. To change one element:

```
x[4,5] <- 5
```

To change a complete column:

```
x <- matrix(rnorm(100),ncol=10)
x[ ,1] <- 1:10
```

2. A logical matrix with the same dimension as `x`

```
x <- matrix(1:36,ncol=6)
y <- x>19
y
      [,1]  [,2]  [,3]  [,4] [,5] [,6]
[1,] FALSE FALSE FALSE FALSE TRUE TRUE
[2,] FALSE FALSE FALSE  TRUE TRUE TRUE
[3,] FALSE FALSE FALSE  TRUE TRUE TRUE
[4,] FALSE FALSE FALSE  TRUE TRUE TRUE
[5,] FALSE FALSE FALSE  TRUE TRUE TRUE
[6,] FALSE FALSE FALSE  TRUE TRUE TRUE

x[y]

[1] 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
```

Note that the result of subscripting with a logical matrix is a vector. This mechanism can be used to replace elements of a matrix. For example:

```
x <- matrix(rnorm(100),ncol=10)
x[x>0] <- 0
```

3. A matrix `r` with two columns A row of r consists of two numbers, each row of r selects a matrix element of `x`. The result is a vector with the selected elements from x.

```
x <- matrix(1:36,ncol=6)
x
[,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    7   13   19   25   31
[2,]    2    8   14   20   26   32
[3,]    3    9   15   21   27   33
[4,]    4   10   16   22   28   34
[5,]    5   11   17   23   29   35
[6,]    6   12   18   24   30   36

r <- cbind( c(1,2,5), c(3,4,4))
r
     [,1] [,2]
[1,]    1    3
[2,]    2    4
[3,]    5    4

x[r]
[1] 13 20 23
```

4. A single number or one vector of numbers. In this case the matrix is treated like a vector where all the columns are stacked.

```
x <- matrix(1:36,ncol=6)
x[3];x[9];x[36]
[1] 3
[1] 9
[1] 36
x[21:30]
[1] 21 22 23 24 25 26 27 28 29 30
```

4.3 Manipulating Data frames

4.3.1 Extracting data from data frames

A data frame can be considered as a generalized matrix, consequently all subscripting methods that work on matrices also work on data frames. However, data frames offer a few extra possibilities. Lets import the data in the file cars.csv, a comma separated text file, so that different aspects of data frame manipulation can be demonstrated.

```
cars <- read.csv("cars.csv", row.names=1)
```

The argument `row.names` is specified in the `read.csv` function because the first column of the data file contains row names that we will use in our data fame. To see the column names of the cars data frame use the function names:

```
names(cars)
[1] "Price"       "Country"     "Reliability" "Mileage"
[5]  "Type"        "Weight"     "Disp."       "HP"
```

To select a specific column from a data frame use the $ symbol or double square brackets and quotes:

```
prices <- cars$Price
prices <- cars[["Price"]]
```

The object `prices` is a vector. If you want the result to be a data frame use single square brackets:

```
prices2 <- cars["Price"]
```

When using single brackets it is possible to select more than one column from a data frame. The result is again a data frame:

```
test <- cars[c("Price","Type")]
```

To select a specific row by name of the data frame 'cars' use the following R code:

```
cars["Nissan Van 4", ]
             Price Country Reliability Mileage Type Weight Disp.  HP
Nissan Van 4 14799   Japan          NA      19  Van   3690   146 106
```

The result is a data frame with one row. To select more rows use a vector of names:

```
cars[c("Nissan Van 4", "Dodge Grand Caravan V6"), ]
```

If the given row name does not exist, R will return a row with `NA`'s.

```
cars["Lada",]
Price Country Reliability Mileage Type Weight Disp. HP
NA    NA      NA          NA      NA   NA     NA    NA NA
```

Rows from a data frame can also be selected using row numbers. Select cases 10 trough 14 from the cars data frame.

```
cars[10:14,]
                    Price   Country Reliability Mileage   Type Weight Disp.  HP
Subaru Justy 3       5866     Japan          NA      34  Small   1900    73  73
Toyota Corolla 4     8748 Japan/USA           5      29  Small   2390    97 102
Toyota Tercel 4      6488     Japan           5      35  Small   2075    89  78
Volkswagen Jetta 4   9995   Germany           3      26  Small   2330   109 100
Chevrolet Camaro V8 11545       USA           1      20 Sporty   3320   305 170
```

The first few rows or the last few rows can be extracted by using the functions `head` or `tail`.

```
head(cars,3)
                Price Country Reliability Mileage  Type Weight Disp.  HP
Eagle Summit 4   8895     USA           4      33 Small   2560    97 113
Ford Escort   4  7402     USA           2      33 Small   2345   114  90
Ford Festiva 4   6319   Korea           4      37 Small   1845    81  63

tail(cars,2)
                Price Country Reliability Mileage Type Weight Disp.  HP
Nissan Axxess 4 13949   Japan          NA      20  Van   3185   146 138
Nissan Van 4    14799   Japan          NA      19  Van   3690   146 106
```

To subset specific cases from a data frame you can also use a logical vector. When you provide a logical vector in a data frame subscript, only the cases which correspond with a `TRUE` are selected. Suppose you want to get all cars from the cars data frame that have a weight of over 3500. First create a logical vector `tmp`:

```
tmp <- cars$Weight > 3500
```

Use this vector to subset:

```
cars[tmp, ]
                          Price Country Reliability Mileage   Type Weight Disp.  HP
       Ford Thunderbird V6 14980     USA           1      23 Medium   3610   232 140
      Chevrolet Caprice V8 14525     USA           1      18  Large   3855   305 170
Ford LTD Crown Victoria V8 17257     USA           3      20  Large   3850   302 150
    Dodge Grand Caravan V6 15395     USA           3      18    Van   3735   202 150
          Ford Aerostar V6 12267     USA           3      18    Van   3665   182 145
              Mazda MPV V6 14944   Japan           5      19    Van   3735   181 150
              Nissan Van 4 14799   Japan          NA      19    Van   3690   146 106
```

A handy alternative is the function `subset`. It returns a the subset as a data frame. The first argument is the data frame. The second argument is a logical expression. In this expression you use the variable names without proceeding them with the name of the data frame, as in the above example.

```
subset(cars, Weight > 3500 & Price < 15000)
                     Price Country Reliability Mileage   Type Weight Disp.  HP
Ford Thunderbird V6  14980     USA           1      23 Medium   3610   232 140
Chevrolet Caprice V8 14525     USA           1      18  Large   3855   305 170
Ford Aerostar V6     12267     USA           3      18    Van   3665   182 145
Mazda MPV V6         14944   Japan           5      19    Van   3735   181 150
Nissan Van 4         14799   Japan          NA      19    Van   3690   146 106
```

4.3.2 Adding columns to a data frame

The function `cbind` can be used to add additional columns to a data frame. For example, the vector '`maxvel`' with the maximum velocities of the cars can be added to the 'cars' data frame as follows.

```
new.cars <- cbind(cars, Max.Vel = maxvel)
```

The left hand side of the = specifies the column name in the 'new.cars' data frame and the right hand side is the vector you want to add. Or alternatively, use the following syntax

```
cars$max.vel = maxvel
```

The function `cbind` can also be used on two or more data frames. For example

```
cbind(dataframe1, dataframe2)
```

4.3.3 Combining data frames

Use the function `rbind` to combine (or stack) two or more data frames. Consider the following two data frames 'rand.df1' and 'rand.df2'.

```
rand.df1 <- data.frame(
    norm = rnorm(5),
    binom = rbinom(5,10,0.1),
    unif=runif(5)
)
rand.df1
        norm binom      unif
1 -1.1477095     2 0.6230449
2  0.6689266     0 0.9921276
3  0.3738174     2 0.7115776
4  2.2641381     2 0.9318150
5 -1.7682772     0 0.6455379

rand.df2 <- data.frame(
    chisq = rchisq(5,3),
    binom = rbinom(5,10,0.1),
    unif=runif(5)
)
```

```
rand.df2
      chisq binom      unif
1  1.955729     1 0.4543552
2 12.661964     1 0.8731595
3  7.433911     1 0.9460346
4  3.642188     0 0.6632598
5  6.134571     1 0.7688208
```

These two data frames have two columns in common: 'binom' and 'unif'. When we only need to combine the common columns of these data frames, you can use the subscripting mechanism and the function `rbind`:

```
rand.comb <- rbind(
    rand.df1[ , c("unif","binom")],
    rand.df2[ , c("unif", "binom")]
)
rand.comb
        unif binom
1  0.6230449     2
2  0.9921276     0
3  0.7115776     2
4  0.9318150     2
5  0.6455379     0
6  0.4543552     1
7  0.8731595     1
8  0.9460346     1
9  0.6632598     0
10 0.7688208     1
```

The functions `rbind` expects that the two data frames have the same columns. The function `rbind.fill` in the 'reshape' package can stack two or more data frames with any columns. It will fill a missing column with `NA`.

```
library(reshape)
rbind.fill(rand.df1,rand.df2,rand.df1)
         norm binom       unif     chisq
1  -3.0309036     1 0.39182298        NA
2   1.5897306     0 0.04189106        NA
3   1.3976871     2 0.09756326        NA
4   0.4867048     0 0.70522637        NA
5  -1.7282814     0 0.42753294        NA
6          NA     0 0.98808959 5.6099156
7          NA     1 0.56966460 2.5105316
8          NA     1 0.53950251 1.0920222
9          NA     0 0.01064824 0.2301267
10         NA     1 0.87821054 3.8488757
```

4.3.4 Merging data frames

Two data frames can be merged into one data frame using the function `merge`. (The join operation in database terminology). If the original data frames contain identical columns, these columns only appear once in the merged data frame. Consider the following two data frames:

```
test1 <- read.delim("test1.txt", sep =" ")
test1
    name year   BA   HR
1   Dick 1963 0.12 0.27
2   Gose 1970 0.53 0.74
3   Rolf 1971 0.53 0.28
4 Heleen 1974 0.81 0.29

test2 <- read.delim("test2.txt", sep =" ")
    name year    A   FA
1   Dick 1963 0.42 0.12
2   Gose 1970 0.26 0.57
3   Rolf 1971 0.87 0.37
4 Heleen 1974 0.86 0.15

test.merge <- merge(test1,test2)
test.merge
 name year   BA   HR    A   FA
1   Dick 1963 0.12 0.27 0.42 0.12
2   Gose 1970 0.53 0.74 0.26 0.57
3 Heleen 1974 0.81 0.29 0.86 0.15
4   Rolf 1971 0.53 0.28 0.87 0.37
```

By default the `merge` function leaves out rows that where not matched, consider the following data sets.

```
quotes = data.frame(date=1:100, quote=runif(100))
testfr = data.frame(date=c(5,7,9, 110), position = c(45,89,14,90))
```

To extend the data frame `testfr` with the wright quote data from the data frame `quotes`, and to keep the last row of `testfr` for which there is no quote use the following code.

```
testfr = merge(quotes,testfr,all.y=TRUE)
testfr
  date     quote position
1    5 0.6488612       45
```

```
2    7 0.4995684       89
3    9 0.5242953       14
4  110        NA       90
```

For more complex examples see the Help file of the function `merge`: `?merge`.

4.3.5 Aggregating data frames

The function `aggregate` is used to aggregate data frames. It splits the data frame into groups and applies a function on each group. The first argument is the data frame, the second argument is a list of grouping variables, the third argument is a function that returns a scalar. A small example:

```
gr <- c("A","A","B","B")
x <- c(1,2,3,4)
y <- c(4,3,2,1)
myf <- data.frame(gr, x, y)
aggregate(myf, list(myf$gr), mean)
  Group.1 gr   x   y
1       A NA 1.5 3.5
2       B NA 3.5 1.5
```

R will apply the function on each column of the data frame. This means also on the grouping column 'gr'. This column is of type factor, numerical calculations can not be performed on factors hence the `NA`'s. You can leave out the grouping columns when calling the `aggregate` function.

```
aggregate(
    myf[, c("x","y")],
    list(myf$gr),
    mean
)
  Group.1   x   y
1       A 1.5 3.5
2       B 3.5 1.5
```

4.3.6 Stacking columns of data frames

The function `stack` can be used to stack columns of a data frame into one column and one grouping column. Consider the following example:

```
group1 <- rnorm(3)
group2 <- rnorm(3)
group3 <- rnorm(3)
df <- data.frame(group1,group2, group3)
stack(df)
       values    ind
1  0.63706989 group1
2 -0.76002786 group1
3  0.05912762 group1
4  0.20074146 group2
5  1.11071470 group2
6  0.43529956 group2
7  1.35128903 group3
8 -0.39660149 group3
9 -0.65003395 group3
```

So by default all the columns of a data frame are stacked. Use the `select` argument to stack only certain columns.

```
stack(df,select=c("group1","group3"))
```

4.3.7 Reshaping data

The function `reshape` can be used to transform a data frame in *wide format* into a data frame in *long format*. In a wide format data frame the different measurements of one 'subject' are in multiple columns, whereas a long format data frame has the different measurements of one subject in multiple rows.

```
df.wide <- data.frame(
   Subject = c(1, 2),
   m1      = c(4, 5),
   m2      = c(5.6, 7.8),
   m3      = c(3.6, 6.7)
)
df.wide
  Subject m1  m2  m3
1       1  4 5.6 3.6
```

```
2       2  5 7.8 6.7

df.long <- reshape(df.wide,
   varying   = list(c("m1", "m2", "m3")),
   idvar     = "Subject",
   direction = "long",
   v.names   = "Measurement"
)
df.long
      Subject time Measurement
1.1       1    1         4.0
2.1       2    1         5.0
1.2       1    2         5.6
2.2       2    2         7.8
1.3       1    3         3.6
2.3       2    3         6.7
```

4.4 Attributes

Vectors, matrices and other objects in general, may have attributes. These are other objects attached to the main object. Use the function `attributes` to get a list of all the attributes of an object.

```
x <- rnorm(10)
attributes(x)
NULL
```

In the above example the vector `x` has no attributes. You can either use the function `attr` or the function `structure` to attach an attribute to an object.

```
attr(x, "description") <- "The unit is month"
x
[1]  1.3453003 -1.4395975  1.0163646 -0.6566600  0.4412399
[6] -1.2427861  1.4967771  0.6230324 -0.5538395  1.0781191
attr(, "description"):
[1] "The unit is month"
```

The first argument of the function `attr` is the object, the second argument is the name of the attribute. The expression on the right hand side of the assignment operator will be the attribute value. Use the `structure` function to attach more than one attribute to an object.

```
x <- structure(x, atr1=8,atr2="test")
x
 [1]  1.3453003 -1.4395975  1.0163646 -0.6566600  0.4412399
 [6] -1.2427861  1.4967771  0.6230324 -0.5538395  1.0781191
attr(, "description"):
[1] "The unit is month"
attr(, "atr1"):
[1] 8
attr(, "atr2"):
[1] "test"
```

When an object is printed, the attributes (if any) are printed as well. To extract an attribute from an object use the functions `attributes` or `attr`. The function `attributes` returns a list of all the attributes from which you can extract a specific component.

```
attributes(x)
$description:
[1] "The unit is month"

$atr1:
[1] 8

$atr2:
[1] "test"
```

In order to get the description attribute of `x` use:

```
attributes(x)$description
[1] "The unit is month"
```

Or type in the following construction:

```
attr(x,"description")
[1] "The unit is month"
```

4.5 Character manipulation

There are several functions in R to manipulate or get information from character objects.

4.5.1 The functions `nchar`, `substring` and `paste`

```
x <- c("a","b","c")
mychar1 <- "This is a test"
mychar2 <- "This is another test"
charvector <- c("a", "b", "c", "test")
```

The function `nchar` returns the length of a character object, for example:

```
nchar(mychar1)
[1] 15
nchar(charvector)
[1] 1 1 1 4
```

The function `substring` returns a substring of a character object. For example:

```
x <- c("Gose", "Longhow", "David")
substring(x,first=2,last=4)
[1] "ose" "ong" "avi"
```

The function `paste` will paste two or more character objects. For example, to create a character vector with: "number.1", "number.2", ...,"number.10" proceed as follows:

```
paste("number",1:10, sep=".")
[1] "number.1"  "number.2"  "number.3"  "number.4"
[5] "number.5"  "number.6"  "number.7"  "number.8"
[9] "number.9"  "number.10"
```

The argument `sep` is used to specify the separating symbol between the two character objects.

```
paste("number",1:10, sep="-")
[1] "number-1"  "number-2"  "number-3"  "number-4"
[5] "number-5"  "number-6"  "number-7"  "number-8"
[9] "number-9"  "number-10"
```

Use `sep=""` for no space between the character objects.

4.5.2 Finding patterns in character objects

The functions `regexpr` and `grep` can be used to find specific character strings in character objects. The functions use so-called regular expressions, a handy format to specify search pattern. See the help for `regexpr` to find out more about regular expressions.

Let's extract the row names from our data frame 'cars'.

```
car.names <- row.names(cars)
```

We want to know if a string in 'car.names' starts with 'Volvo' and if there is, the position it has in 'car.names'. Use the function `grep` as follows:

```
grep("Volvo", car.names)
[1] 37
```

So element 37 of the `car.names` vector is a name that contains the string 'Volvo', which is confirmed by a quick check:

```
car.names[37]
[1] "Volvo 240 4"
```

To find the car names with second letter 'a', we must use a more complicated regular expression

```
tmp <- grep("^.a",car.names)
car.names[tmp]
[1] "Eagle Summit 4"    "Mazda Protege 4"
[3] "Mazda 626 4"       "Eagle Premier V6"
[5] "Mazda 929 V6"      "Mazda MPV V6"
```

For those who are familiar with wildcards (aka globbing) there is a handy function `glob2rx` that transforms a wildcard to a regular expression.

```
rg <- glob2rx("*.tmp")
rg
[1] "^.*\\.tmp$"
```

To find patterns in texts you can also use the `regexpr` function. This function also makes use of regular expressions, however it returns more information than `grep`.

```
Volvo.match <- regexpr("Volvo",car.names)
Volvo.match
 [1] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[19] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[37]  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[55] -1 -1 -1 -1 -1 -1
attr(, "match.length"):
 [1] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[19] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[37]  5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[55] -1 -1 -1 -1 -1 -1
```

The result of `regexpr` is a numeric vector with a `match.length` attribute. A minus one means no match was found, a positive number means a match was found. In our example we see that element 37 of 'Volvo.match' equals one, which means that 'Volvo' is part of the character string in element 37 of 'car.names'. Again a quick check:

```
car.names[37]
[1] "Volvo 240 4"
```

In the above result you could immediately see that element 37 of 'car.names' is a match. If character vectors become too long to see the match quickly, use the following trick:

```
index <- 1:length(car.names)
index[Volvo.match > 0]
[1] 37
```

The result of the function `regexpr` contains the attribute match.length, which gives the length of the matched text. In the above example match Volvo consists of 5 characters. This attribute can be used together with the function `substring` to extract the found pattern from the character object.

Consider the following example which uses a regular expression, the 'match.length' attribute, and the function `substring` to extract the numeric part and character part of a character vector.

```
x <- c("10 Sept", "Oct 9th", "Jan 2", "4th of July")
w <- regexpr("[0-9]+", x)
```

The regular expression `"[0-9]+"` matches an integer.

```
w
[1] 1 5 5 1
attr(, "match.length"):
[1] 2 1 1 1

# The 1 means there is a match on position 1 of "10 Sept"
# The 5 means there is a match on position 5 of "Oct 9th"
# The 5 means there is a match on position 5 of "Jan 2"
# The 1 means there is a match on position 1 of "4th of July"
```

In the attribute `match.length` the 2 indicates the length of the match in "10 Sept".

Use the `substring` function to extract the integers. Note that the result of the substring function is of type character. To convert that to numeric, use the `as.numeric` function:

```
as.numeric(substring(x, w, w+attr(w, "match.length")-1))
[1] 10  9  2  4
```

4.5.3 Replacing characters

The functions `sub` and `gsub` are used to replace a certain pattern in a character object with another pattern.

```
mychar <- c("My_test", "My_Test_3", "_qwerty_pop_")
sub(pattern="[_]", replacement=".", x=mychar)
[1] "My.test"      "My.Test_3"    ".qwerty_pop_"
gsub(pattern="[_]", replacement=".", x=mychar)
[1] "My.test"      "My.Test.3"    ".qwerty.pop."
```

Note that by default, the `pattern` argument is a regular expression. When you want to replace a certain string it may be handy to use the `fixed` argument as well.

```
mychar <- c("mytest", "abctestabc", "test.po.test")
gsub(pattern="test", replacement="", x=mychar, fixed=TRUE)
[1] "my"     "abcabc" ".po."
```

4.5.4 Splitting characters

A character string can be split using the function `strsplit`. The two main arguments are `x` and `split`. The function returns the split results in a list, each list componenent is the split result of an element of `x`.

```
strsplit(x = c("Some text", "another string", split = NULL)
[[1]]
[1] "S" "o" "m" "e" " " "t" "e" "x" "t"

[[2]]
 [1] "a" "n" "o" "t" "h" "e" "r" " " "s" "t" "r" "i" "n" "g"
```

The argument `x` is a vector of characters, and `split` is a character vector containing regular expressions that are used for the split. If it is NULL as in the above example, the character strings are split into single characters. If it is not null, R will look at the elements in `x`, if the split string can be matched the characters left of the match will be in the output and the characters right of the match will be in the output.

```
strsplit(
   x = c("Some~text" , "another-string", "Amsterdam is a nice city"),
   split = "[~-]"
)
[[1]]
[1] "Some" "text"

[[2]]
[1] "another" "string"

[[3]]
[1] "Amsterdam is a nice city"
```

4.6 Creating factors from continuous data

The function `cut` can be used to create factor variables from continuous variables. The first argument `x` is the continuous vector and the second argument `breaks` is a vector of breakpoints, specifying intervals. For each element in `x` the function `cut` returns the interval as specified by `breaks` that contains the element.

```
x <- 1:15
breaks <- c(0,5,10,15,20)
cut(x,breaks)
 [1] (0,5]   (0,5]   (0,5]   (0,5]   (0,5]   (5,10]  (5,10]  (5,10]  (5,10]
[10] (5,10]  (10,15] (10,15] (10,15] (10,15] (10,15]
Levels: (0,5] (5,10] (10,15] (15,20]
```

The function `cut` returns a vector of tye 'factor', each element of this vector shows the interval to which the corresponding element of the original vector corresponds. If only one number is specified for the argument `breaks`, that number is used to divide `x` into equal length intervals.

```
cut( x, breaks=5)
 [1] (0.986,3.79] (0.986,3.79] (0.986,3.79] (3.79,6.6]   (3.79,6.6]
 [6] (3.79,6.6]   (6.6,9.4]    (6.6,9.4]    (6.6,9.4]    (9.4,12.2]
[11] (9.4,12.2]   (9.4,12.2]   (12.2,15]    (12.2,15]    (12.2,15]
Levels: (0.986,3.79] (3.79,6.6] (6.6,9.4] (9.4,12.2] (12.2,15]
```

The names of the different levels are created by R automatically, they have the form (a,b]. You can change this by specifying an extra `labels` argument.

```
x <- rnorm(15)
cut(x, breaks=3, labels=c("low","medium","high"))
 [1] high   medium medium medium medium high   low    high   low    low
[11] high   low    low    medium high
Levels: low medium high
```

5 Writing functions

5.1 Introduction

Most tasks are performed by calling a function in R. In fact, everything we have done so far is calling an existing function which then performed a certain task resulting in some kind of output. A function can be regarded as a collection of statements and is an object in R of class 'function'. One of the strengths of R is the ability to extend R by writing new functions. The general form of a function is given by:

```
functionname <- function(arg1, arg2,...) {
  Body of function: a collection of valid statements
}
```

In the above display `arg1` and `arg2` in the function header are input arguments of the function. Note that a function doesn't need to have any input arguments. The body of the function consists of valid R statements. For example, the commands, functions and expressions you type in the R console window. Normally, the last statement of the function body will be the return value of the function. This can be a vector, a matrix or any other data structure.

The following short function `meank` calculates the mean of a vector `x` by removing the `k` percent smallest and the `k` percent largest elements of the vector.

```
meank <- function(x,k){
   xt <- quantile(x, c(k,1-k))
   mean( x[ x > xt[1] & x < xt[2] ])
}
```

Once the function has been created, it can be ran.

```
test <- rnorm(100)
meank(test)
[1] 0.00175423
```

The function `meank` calls two standard functions, `quantile` and `mean`. Once `meank` is created it can be called from any other function.

If you write a short function, a one-liner or two-liner, you can type the function directly in the console window. If you write longer functions, it is more convenient to use a script file. Type the function definition in a script file and run the script file. Note that when you run a script file with a function definition, you will only define the function (you will create a new object). To actually run it, you will need to call the function with the necessary arguments.

You can use your favorite text editor to create or edit functions. Use the function `source` to evaluate expressions from a file. Suppose 'meank.txt' is a text file, saved on your hard disk, containing the function definition of `meank`.

```
meank <- function(x,k){
   xt <- quantile(x,c(k,1-k))
   mean(x[ x>xt[1] & x<xt[2] ])
}
```

The following statement will create the function `meank` in R:

```
# note the use of double slashes...
source("C:\\SFunctions\\meank.txt")
```

Now you can run the function:

```
meank(test)
[1] 0.00175423
```

If you want to put a comment inside a function, use the `#` symbol. Anything between the `#` symbol and the end of the line will be ignored.

```
test <- function(x){

# This line will be ignored
# It is useful to insert code explanations for others (and yourself!)

  sqrt(2*x)
}
```

Writing large functions in R can be difficult for novice users. You may wonder where and how to begin, how to check input parameters or how to use loop structures.

Fortunately, the code of many functions can be viewed directly. For example, just type the name of a function without brackets in the console window and you will get the code. Don't be intimidated by the (lengthy) code. Learn from it, by trying to read line by line and looking at the help of the functions that you don't know yet. Some functions call 'internal' functions or pre-compiled code, which can be recognized by calls like: `.C`, `.Internal` or `.Call`.

5.2 Arguments and variables

5.2.1 Required and optional arguments

When calling functions in R, the syntax of the function definition determines whether argument values are required or optional. With optional arguments, the specification of the arguments in the function header is:

```
argname = defaultvalue
```

In the following function, for example, the argument `x` is required and R will give an error if you don't provide it. The argument `k` is optional, having the default value 2:

```
power <- function(x, k=2){
  x^k
}

power(5)
[1]  25

power()
Error in power() : argument "x" is missing, with no default
```

However, we can specify a different value for `k`:

```
power(5,3)
[1]  125
```

5.2.2 The '...' argument

The three dots argument can be used to pass arguments from one function to another. For example, graphical parameters that are passed to plotting functions or numerical parameters that are passed to numerical routines. Suppose you write a small function to plot the sin function from zero to `xup`.

```
plotsin <- function(xup = 2*pi, ...)
{
  x <- seq(0, xup, l = 100)
  plot(x,sin(x), type = "l", ...)
}

plotsin(col="red")
```

The function `plotsin` now accepts any argument that can be passed to the `plot` function (like `col`, `xlab`, etc.) without needing to specify those arguments in the header of `plotsin`.

5.2.3 Local variables

Assignments of variables inside a function are local, unless you explicitly use a global assignment (the `<<-` construction or the `assign` function). This means a normal assignment within a function will not overwrite objects outside the function. An object created within a function will be lost when the function has finished. Only if the last line of the function definition is an assignment, then the result of that assignment will be returned by the function.

In the next example an object `x` will be defined with value zero. Inside the function `functionx`, xis defined with value 3. Executing the function `functionx` will not affect the value of the global variable 'x'.

```
x <- 0
functionx <- function(){
  x <- 3
}

functionx()
[1]  3
 x
[1]  0
```

If you want to change the global variable `x` with the return value of the function `functionx`, you can assign the function result to `x`.

```
# overwriting the object x with the result of functionx
x <- functionx()
```

The arguments of a function can be objects of any type, even functions! Consider the next example:

```
test <- function(n, fun)
{
  u <- runif(n)
  fun(u)
}

test(10,sin)
[1] 0.28078332 0.30438298 0.55219120 0.37357375 ...
```

The second argument of the function `test` needs to be a function which will be called inside the function.

5.2.4 Returning an object

Often the purpose of a function is to do some calculations on input arguments and return the result. By default the last expression of the function will be returned.

```
myf <- function(x,y){
  z1 <- sin(x)
  z2 <- cos(y)
  z1+z2
}
```

In the above example `z1 + z2` is returned, note that the individual objects `z1` and `z2` will be lost. You can only return one object. If you want to return more than one object, you have to return a list where the components of the list are the objects to be returned. For example:

```
myf <- function(x,y){
  z1 <- sin(x)
  z2 <- cos(y)
  list(z1,z2)
}
```

To exit a function before it reaches the last line, use the `return` function. Any code after the return statement inside a function will be ignored. For example:

```
myf <- function(x,y){
  z1 <- sin(x)
  z2 <- cos(y)
  if(z1 < 0){
    return( list(z1,z2))
  else{
    return( z1+z2)
  }
}
```

5.2.5 The Scoping rules

The scoping rules of a programming language are the rules that determine how the programming language finds a value for a variable. This is especially important for *free* variables inside a function and for functions defined inside a function. Let's look at the following example function.

```
myf <- function(x)
{
   y = 6
   z = x + y + a1
   a2 = 9
   insidef = function(p){
      tmp = p*a2
      sin(tmp)
   }
   2*insidef(z)
}
```

In the above function

- `x, p` are formal arguments.
- `y, tmp` are local variables.
- `a2` is a local variable in the function `myf`.
- `a2` is a free variable in the function `insidef`.

R uses a so-called *lexical scoping* rule to find the value of free variables, see [3]. With lexical scoping, free variables are first resolved in the environment in which the function was created. The following calls to the function `myf` show this rule.

```
## R tries to find a1 in the environment where myf was created
## but there is no object a1
myf(8)
Error in myf(8) : object "a1" not found

## define the objects a1 and a2 but what value
## did a2 in the function insidef get?
a1 <- 10
a2 <- 1000
myf(8)
[1] 1.392117

## It took a2 in myf, so a2 has the value 9
```

5.2.6 Lazy evaluation

When writing functions in R, a function argument can be defined as an expression like:

```
myf <- function(x, nc = length(x))
{
   rest of the function
}
```

When arguments are defined in such a way you must be aware of the *lazy evaluation* mechanism in R. This means that arguments of a function are not evaluated until needed. Consider the following examples.

```
myf <- function(x, nc = length(x))
{
   x <- c(x, x)
   print(nc)
}
xin <- 1:10
myf(xin)
[1] 20
```

The argument `nc` is evaluated after `x` has doubled in length, it is not ten, the initial length of `x` when it entered the function.

```
logplot <- function(y, ylab = deparse(substitute(y))) {
   y <- log(y)
   plot(y, ylab = ylab)
}
```

The plot will create a nasty label on the y axis. This is the result of lazy evaluation, `ylab` is evaluated after `y` has changed. One solution is to force an evaluation of `ylab` first.

```
logplot <- function(y, ylab = deparse(substitute(y))) {
   ylab
   y <- log(y)
   plot(y, ylab = ylab)
}
```

5.3 Control flow

The following shows a list of constructions to perform testing and looping. These constructions can also be used outside a function to control the flow of execution.

5.3.1 Tests with `if` and `switch`

The general form of the if construction has the form

```
if(test)
{
   ...true statements...
}
else
{
   ...false statements...
}
```

where `test` is a logical expression like `x < 0`, `x < 0 & x > -8`. R evaluates the logical expression if it results in `TRUE` then it executes the `true statements`. If the logical expression results in `FALSE` then it executes the `false statements`. Note that it is not neccesary to have the `else` block.

Simple example Adding two vectors in R of different length will cause R to recycle the shorter vector. The following function adds the two vectors by chopping of the longer vector so that it has the same length as the shorter.

```
myplus <- function(x, y){
  n1 <- length(x)
  n2 <- length(y)
  if(n1 > n2){
    z <- x[1:n2] + y
  }
  else{
    z <- x + y[1:n1]
  }
  z
}
myplus(1:10, 1:3)
[1] 2 4 6
```

The `switch` function has the following general form.

```
switch(object,
    "value1" = {expr1},
    "value2" = {expr2},
    "value3" = {expr3},
               {other expressions}
)
```

If `object` has value `value1` then `expr1` is executed, if it has `value2` then `expr2` is executed and so on. If `object` has no match then `other expressions` is executed. Note that the block {`other expressions`} does not have to be present, the switch will return NULL in case `object` does not match any value. An expression `expr1` in the above construction can consist of multiple statements. Each statement should be separated with a `;` or on a separate line and surrounded by curly brackets.

Simple example Choosing between two calculation methods.

```
mycalc <- function(x, method="ml"){
  switch(method,
    "ml"  = { my.mlmethod(x) },
    "rml" = { my.rmlmethod(x) }
  )
}
```

5.3.2 Looping with `for`, `while` and `repeat`

The `for`, `while` and `repeat` constructions are designed to perfom loops in R. They have the following forms.

```
for (i in for_object)
{
  some expressions
}
```

In the for loop `some expressions` are evaluated for each element `i` in `for_object`.

Simple example A recursive filter.

```
arsim <- function(x, phi){
  for(i in 2:length(x))
  {
    x[i] <- x[i] + phi*x[i-1]
  }
  x
}
arsim(1:10, 0.75)
 [1]  1.000000  2.750000  5.062500  7.796875 10.847656
 [6] 14.135742 17.601807 21.201355 24.901016 28.675762
```

Note that the `for_object` could be a vector, a matrix, a data frame or a list.

```
while (condition)
{
   some expressions
}
```

In the while loop `some expressions` are repeatedly executed until the logical `condition` is `FALSE`. Make sure that the condition is `FALSE` at some stage, otherwise the loop will go on indefinitely.

Simple example

```
mycalc <- function(){
  tmp <- 0
  n <- 0
  while(tmp < 100){
    tmp <- tmp + rbinom(1,10,0.5)
    n <- n +1
  }
  cat("It took ")
  cat(n)
  cat(" iterations to finish \n")
}
```

```
repeat
{
   some expressions
}
```

In the repeat loop `some expressions` are repeated 'infinitely', so `repeat` loops will have to contain a `break` statement to escape them.

5.4 Debugging your R functions

5.4.1 The `traceback` function

The R language provide the user with some tools to track down *unexpected behavior* during the execution of (user written) functions. For example,

- A function may throw *warnings* at you. Although warnings do not stop the execution of a function and could be ignored, you should check out why a warning is produced.
- A function stops because of an error. Now you must really fix the function if you want it to continue to run until the end.
- Your function runs without warnings and errors, however the number it returns does not make any sense.

The first thing you can do when an error occurs is to call the function `traceback`. It will list the functions that were called before the error occurred. Consider the following two functions.

```
myf <- function(z)
{
  x <- log(z)
  if( x > 0)
  {
    print("PPP")
  }
  else
  {
    print("QQQ")
  }
}

testf <- function(pp)
{
  myf(pp)
}
```

Executing the command `testf(-9)` will result in an error, execute traceback to see the function calls before the error.

```
Error in if (x > 0) { : missing value where TRUE/FALSE needed
In addition: Warning message:
NaNs produced in: log(x)

traceback()
2: myf(pp)
1: testf(-9)
```

Sometimes it may not be obvious where a warning is produced, in that case you may set the option

```
options(warn = 2)
```

Instead of continuing the execution, R will now halt the execution if it encounters a warning.

5.4.2 The `warning` and `stop` functions

You, as the writer of a function, can also produce errors and warnings. In addition to putting ordinary print statements like `print("Some message")` in your function, you can use the function `warning`. For example,

```
variation <- function(x)
{
  if(min(x) <= 0)
  {
    warning("variation only useful for positive data")
  }
  sd(x)/mean(x)
}

variation(rnorm(100)
[1] 19.4427
Warning message:
variation only useful for positive data in: variation(rnorm(100))
```

If you want to raise an error you can use the function `stop`. In the above example when we replace `warning` by `stop` R would halt the execution.

```
variation(rnorm(100))
Error in variation(rnorm(100)) : variation only useful for positive data
```

R will treat your warnings and errors as normal R warnings and errors. That means for example, the function `traceback` can be used to see the call stack when an error occurred.

5.4.3 Stepping through a function

With traceback you will now in which function the error occurred, it will not tell you where in the function the error occurred. To find the error in the function you can use the function `debug`, which will tell R to execute the function in debug mode. If you want to step through *everything* you will need to set debug flag for the main function and the functions that the main function calls:

```
debug(testf)
debug(myf)
```

Now execute the function `testf`, R will display the body of the function and a browser environment is started.

```
testf(-9)
debugging in: testf(-9)
debug: {
    myf(pp)
}
Browse[1]>
```

In the browser environment there are a couple of special commands you can give.

- `n`, executes the current line and prints the next one.
- `c`, executes the rest of the function without stopping.
- `Q`, quits the debugging completely, so halting the execution and leaving the browser environment.
- `where`, shows you where you are in the function call stack.

In addition to these special commands, the browser environment acts like an interactive R session, that means you could enter commands like

- `ls()`, show all objects in the local environment, the current function.
- `print(object)` or just `object`, prints the value of the object.
- `675/98876`, just some calculations.
- `object <- 89`, assigning a new value to an object, the debugging process will continue with this new value.

If the debug process is finished remove the debug flag `undebug(myf)`.

5.4.4 The `browser` function

It may happen that an error occurs at the end of a lengthy function. To avoid stepping through the function line by line manually, the function `browser` can be used. Inside your function insert the browser() statement at a location where you want to enter the debugging environment.

```
myf <- function(x)
{
... some code ...
  browser()
  ... some code ...
}
```

Run the function `myf` as normally. When R reaches the `browser()` statement then the normal execution is halted and the debug environment is started.

6 Efficient calculations

6.1 Vectorized computations

The efficiency of calculations depends on how you perform them. Vectorized calculations, for example, avoid going trough individual vector or matrix elements and avoid `for` loops. Though very efficient, vectorized calculations cannot always be used. On the other hand, users having a Pascal or C programming background often forget to apply vectorized calculations where they could be used. We therefore give a few examples to demonstrate its use.

A weighted average Take advantage of the fact that most calculations and mathematical operations already act on each element of a matrix or vector. For example, `log(x)`, `sin(x)` calculate the log and sin on all elements of the vector `x`.

For example, to calculate a weighted average W

$$W = \frac{\sum_i x_i w_i}{\sum_i w_i}$$

in R of the numbers in a vector `x` with corresponding weights in the vector `w`, use:

```
ave.w <- sum(x*w)/sum(w)
```

The multiplication and divide operator act on the corresponding vector elements.

Replacing numbers Suppose we want to replace all elements of a vector which are larger than one by the the value 1. You could use the following construction (as in C or Fortran)

```
## timing the calculation using Sys.time
tmp <- Sys.time()
x <- rnorm(15000)
for (i in 1:length(x)){
  if(x[i] > 1){
```

```
    x[i] <- 1
  }
}
Sys.time - tmp
Time difference of 0.2110000 secs
```

However, the following construction is much more efficient:

```
tmp <- Sys.time()
x <- rnorm(15000)
x[x>1] <- 1
Sys.time() - tmp
Time difference of 0.0400002 secs
```

The second construction works on the complete vector `x` at once instead of going through each separate element. Note that it is more reliable to time an R expression using the function `system.time` or `proc.time`. See their help files.

The `ifelse` function Suppose we want to replace the positive elements in a vector by 1 and the negative elements by -1. When a normal 'if- else' construction is used, then each element must be used individually.

```
tmp <- Sys.time()
x <- rnorm(15000)
for (i in 1:length(x)){
  if(x[i] > 1){
    x[i] <- 1
  }
  else{
    x[i] <- -1
  }
}
Sys.time() - tmp
Time difference of 0.3009999 secs
```

In this case the function `ifelse` is more efficient.

```
tmp <- Sys.time()
x <- rnorm(15000)
x <- ifelse(x>1,1,-1)
tmp - Sys.time()
Time difference of 0.02999997 secs
```

The function `ifelse` has three arguments. The first is a test (a logical expression), the second is the value given to those elements of `x` which pass the test, and the third argument is the value given to those elements which fail the test.

The `cumsum` function To calculate cumulative sums of vector elements use the function cumsum. For example:

```
x <- 1:10
y <- cumsum(x)
y
[1]  1  3  6 10 15 21 28 36 45 55
```

The function `cumsum` also works on matrices in which case the cumulative sums are calculated per column. Use `cumprod` for cumulative products, `cummin` for cumulative minimums and `cummax` for cumulative maximums.

Matrix multiplication In R a matrix-multiplication is performed by the operator `%*%`. This can sometimes be used to avoid explicit looping. An m by n matrix A can be multiplied by an n by k matrix B in the following manner:

```
C <- A %*% B
```

So element C[i,j] of the matrix C is given by the formula:

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

If we choose the elements of the matrices A and B 'cleverly' explicit for-loops could be avoided. For example, column-averages of a matrix. Suppose we want to calculate the average of each column of a matrix. Proceed as follows:

```
A <- matrix(rnorm(1000),ncol=10)
n <- dim(A)[1]
mat.means <- t(A) %*% rep(1/n, n)
```

6.2 The apply and outer functions

6.2.1 the `apply` function

This function is used to perform calculations on parts of arrays. Specifically, calculations on rows and columns of matrices, or on columns of a data frame.

To calculate the means of all columns in a matrix, use the following syntax:

```
M <- matrix(rnorm(10000),ncol=100)
apply(M,1,mean)
```

The first argument of `apply` is the matrix, the second argument is either a 1 or a 2. If one chooses 1 then the mean of each column will be calculated, if one chooses 2 then the mean will be calculated for each row. The third argument is the name of a function that will be applied to the columns or rows.

The function `apply` can also be used with a function that you have written yourself. Extra arguments to your function must now be passed trough the `apply` function. The following construction calculates the number of entries that is larger than a threshold `d` for each column in a matrix.

```
tresh <- function(x,d){
  sum(x>d)
}

M <- matrix(rnorm(10000),ncol=100)
apply(M,1,tresh,0.6)
 [1] 24 26 24 26 31 26 30 27 28 29 26 23 33 23 27 23 27 31 22
[20] 28 25 28 30 25 28 32 23 24 27 33 29 25 26 20 31 28 29 31
[39] 37 36 26 23 23 28 26 28 30 25 23 30 20 34 29 32 34 30 29
[58] 30 37 28 22 27 20 30 24 29 21 26 26 31 26 18 26 34 29 20
[77] 18 27 28 33 33 25 21 35 25 33 27 28 20 35 23 31 25 29 20
[96] 30 27 28 21 31
```

6.2.2 the `lapply` and `sapply` functions

These functions are suitable for performing calculations on the components of a list. Specifically, calculations on the columns of a data frame. If, for instance, you want to find out which columns of the data frame `cars` are of type numeric then proceed as follows:

```
lapply(cars, is.numeric)
$Price:
[1] TRUE

$Country:
[1] FALSE

$Reliability:
[1] FALSE
```

```
$Mileage:
[1] TRUE
...
...
```

The function `sapply` can be used as well:

```
sapply(car.test.frame, is.numeric)
 Price Country Reliability Mileage Type Weight Disp. HP
     T       F           F       T    F      T     T  T
```

The function `sapply` can be considered as the 'simplified' version of `lapply`. The function `lapply` returns a list and `sapply` a vector (if possible). In both cases the first argument is a list (or data frame) , the second argument is the name of a function. Extra arguments that normally are passed to the function should be given as arguments of `lapply` or `sapply`.

```
mysummary <- function(x){
  if(is.numeric(x))
    return(mean(x))
  else
    return(NA)
}

sapply(car.test.frame,mysummary)
     Price Country Reliability  Mileage Type   Weight  Disp.   HP
 12615.67      NA          NA 24.58333   NA 2900.833 152.05 122.35
```

Some attention should be paid to the situation where the output of the function to be called in sapply is not constant. For instance, if the length of the output-vector depends on a certain calculation:

```
myf <- function(x){
  n<-as.integer(sum(x))
  out <- 1:n
  out
}

testdf <- as.data.frame(matrix(runif(25),ncol=5))
sapply(testdf,myf)
$X.1:
[1] 1 2
```

```
$X.2:
[1] 1 0

$X.3:
[1] 1 2 3

$X.4:
[1] 1 2

$X.5:
[1] 1
```

The result will then be an object with a list structure.

6.2.3 The `tapply` function

This function is used to run another function on the cells of a so called *ragged array*. A ragged array is a pair of two vectors of the same size. One of them contains data and the other contains grouping information. The following data vector `x` en grouping vector `y` form an example of a ragged array.

```
x <- rnorm(50)
y <- as.factor(sample(c("A","B","C","D"), size=50, replace=T))
```

A cell of a ragged array are those data points from the data vector that have the same label in the grouping vector. The function `tapply` calculates a function on each cell of a ragged array.

```
tapply(x, y, mean, trim = 0.3)
          A           B          C           D
 -0.4492093 -0.1506878 0.4427229 -0.1265299
```

Combining `lapply` and `tapply` To calculate the mean per group in every column of a data frame, one can use sapply/lapply in combination with `tapply`. Suppose we want to calculate the mean per group of every column in the data frame cars, then we can use the following code:

```
mymean <- function(x,y){
tapply(x,y,mean)
}

lapply(cars, mymean, cars$Country)
$Price
   France   Germany     Japan Japan/USA     Korea    Mexico    Sweden       USA
15930.000 14447.500 13938.053 10067.571  7857.333  8672.000 18450.000 12543.269

$Country
   France   Germany     Japan Japan/USA     Korea    Mexico    Sweden       USA
       NA        NA        NA        NA        NA        NA        NA        NA

$Reliability
   France   Germany     Japan Japan/USA     Korea    Mexico    Sweden       USA
       NA        NA        NA  4.857143        NA  4.000000  3.000000        NA
...
```

6.2.4 The by function

The by function applies a function on parts of a data.frame. Lets look at the cars data again, suppose we want to fit the linear regression model Price Weight for each type of car. First we write a small function that fits the model Price Weight for a data frame.

```
myregr <- function(data)
{
   lm(Price ~ Weight, data = data)
}
```

This function is then passed to the by function

```
outreg <- by(cars, cars$Type, FUN=myregr)
outreg

cars$Type: Compact

Call:
lm(formula = Price ~ Weight, data = data)

Coefficients:
(Intercept)       Weight
   2254.765        3.757

------------------------------------------------------------
```

```
cars$Type: Large

Call:
lm(formula = Price ~ Weight, data = data)

Coefficients:
(Intercept)       Weight
 17881.2839      -0.5183
...
...
```

The output object `outreg` of the `by` function contains all the separate regressions, it is a so called 'by' object. Individual regression objects can be accessed by treating the 'by' object as a list

```
outreg[[1]]
Call:
lm(formula = Price ~ Weight, data = data)

Coefficients:
(Intercept)       Weight
   2254.765        3.757
```

6.2.5 The `outer` function

The function `outer` performs an outer-product given two arrays (vectors). This can be especially useful for evaluating a function on a grid without explicit looping. The function has at least three input-arguments: two vectors `x` and `y` and the name of a function that needs two or more arguments for input. For every combination of the vector elements of `x` and `y` this function is evaluated. Some examples are given by the code below.

```
x <- 1:3
y <- 1:3
z <- outer(x,y,FUN="-")
z
     [,1] [,2] [,3]
[1,]    0   -1   -2
[2,]    1    0   -1
[3,]    2    1    0

x <- c("A", "B", "C", "D")
y <- 1:9
```

```
z <- outer(x, y, paste, sep = "")
z
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,] "A1" "A2" "A3" "A4" "A5" "A6" "A7" "A8" "A9"
[2,] "B1" "B2" "B3" "B4" "B5" "B6" "B7" "B8" "B9"
[3,] "C1" "C2" "C3" "C4" "C5" "C6" "C7" "C8" "C9"
[4,] "D1" "D2" "D3" "D4" "D5" "D6" "D7" "D8" "D9"

x <- seq(-4,4,l=50)
y <- x
myf <- function(x,y){
   sin(x)+cos(y)
}
z <- outer(x,y, FUN = myf)
persp(x,y,z, theta=45, phi=45, shade = 0.45)
```

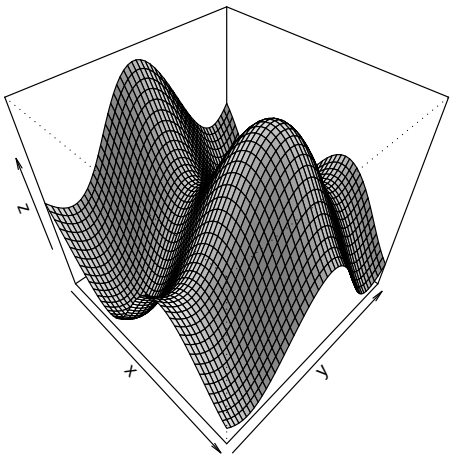

Figure 6.1: A surface plot created with the function `persp`

6.3 Using Compiled code

Sometimes the use of explicit for loops cannot be avoided. When these loops form a real bottleneck in computation time, you should consider implementing these loops in C (or Fortran) and link them to R. This feature is used a lot within the existing R functions

already. In fact, the source code of R is available so you can see many examples. There are a couple of ways to link C or Fortran code to R. On Windows platforms the use of dynamic link libraries (dll's) is probably the easiest solution. For a detailed description see for example, the R manual 'Writing R Extensions' or Chapter 6 and Appendix A of [4].

Reasons to use compiled code

Compiled C or Fortran code is faster than interpreted R code. Loops and especially recursive functions run a lot faster and a lot more efficiently when they are programmed in C or Fortran. It is also possible that you already have some (tested) code at hand that performs a certain routine. Translating the entire C code to R can be cumbersome, so that it may pay off to organize the C code in such a way that it can be used within R.

6.3.1 The `.C` and `.Fortran` interfaces

The `.C` and `.Fortran` interfaces are basic interfaces to C and Fortran. To call a C function that is loaded in R, use the function `.C`, giving it the name of the function (as a character string) and one argument for each argument of the C function. Note that if you pass a vector x to the C code, you also need to explicitly pass its length. In C it is not possible (like `length(x)` in R) to find out the length from only the vector `x`.

```
.C("arsim", x = as.double(x), n = as.integer(length(x)))
```

We'll define the C routine `arsim` in the examples section

To return results to R, modify one or more input arguments of the C function. The value of the `.C()` function is a list with each component matching one argument to the C function. If you name these arguments, as we did in the preceding example, the return list has named components. Your R function can use the returned list for further computations or to construct its own return value, which generally omits those arguments, which are not altered by the C code. Thus, if we wanted to just use the returned value of x, we could call .C() as follows:

```
.C("arsim", x = as.double(x),n = as.integer(length(x)))$x.
```

All arguments of the C routine called via `.C()` must be pointers. All such routines should be `void` functions; if the routine does return a value, it could cause R to crash. R has many classes that are not immediately representable in C. To simplify the interface between R and C, the types of data that R can pass to C code are restricted to the following classes:

- single, integer
- double, complex
- logical, character
- raw, list

The following table shows the correspondence between R data types and C types.

R data type	C data type
`logical`	`long*`
`integer`	`long*`
`double`	`double*`
`complex`	`Rcomplex*`
`character`	`char**`
`raw`	`char*`

6.3.2 The `.Call` and `.External` interfaces

The `.Call` and `.External` interfaces are powerful interfaces that allow you to manipulate R objects from C code and evaluate R expressions from within your C code. It is more efficient than the standard `.C` interface, but because it allows you to work directly with R objects, without the usual R protection mechanisms, you must be careful when programming with it to avoid memory faults and corrupted data.

The `.Call` interface provides you with several capabilities that the standard `.C` interface lacks, including the following

- the ability to create variable-length output variables, as opposed to the pre-allocated objects the `.C` interface expects to write to.
- a simpler mechanism for evaluating R expressions within C.
- the ability to establish direct correspondence between C pointers and R objects.

6.4 Some Compiled Code examples

6.4.1 The arsim example

To apply a first-order recursive filter to a data vector one could write the following R function:

```
arsimR <- function(x,phi){
  n <- length(x)
  if(n > 1){
    for(i in 2:n){
     x[i] <- phi*x[i-1]+x[i]
    }
  }
  x
}

tmp <- Sys.time()
out1 <- arsimR(rnorm(10000), phi = 0.75)
Sys.time() - tmp
Time difference of 0.25 secs
```

We cannot avoid explicit looping in this case, the R function could be slow for large vectors. We implement the function in C and link it to R. In C we can program the `arsim` function and compile it to a dll as follows. First, create a text file arsim.c and insert the following code:

```
void arsim(double *x, long*n, double *phi)
{
  long i;
  for(i=1; i<*n; i++)
    x[i] = *phi * x[i-1] + x[i];
}
```

Then, create a module definition file arsim.def and insert the following text:

```
LIBRARY arsim

EXPORTS
  arsim
```

This module definition file tells which functions are to be exported by the dll. Now compile the two files to a dll. There are many (free and commercial) compilers that can create a dll:

- The GNU compiler collection (free) (http://www.mingw.org)
- lcc (free) (http://www.cs.virginia.edu/ lcc-win32)
- Borland (compiler is free, the IDE is commercial)
- Microsoft Visual studio (commercial)

Lets use `lcc` to create the dll, open a DOS box and type in the following

```
lcc arsim.c
lcclink -dll -nounderscores arsim.obj arsim.def
```

The compiler created the file arsim.dll that can now be linked to R. In R type the following code:

```
mydll = "C:\\DLLLocation\\arsim.dll"
dyn.load(mydll)
is.loaded("arsim")
TRUE
```

The dll is now linked to R and we can use the `.C` interface function to call the `arsim` C function. For convenience, we write a wrapper function `arsimC` that calls the `.C` function

```
arsimC <- function(x, phi)
{
  # only return the first component of the list
  # because the C function only modifies x
  .C("arsim",
    as.numeric(x),
    length(x),
    as.numeric(phi)
  )[[1]]
}

tmp <- Sys.time()
arsimC(rnorm(10000), phi = 0.75)
Sys.time() - tmp
Time difference of 0.00999999 secs
```

As we can see the C code is much faster than the R code, the following graph also shows that.

6.4.2 Using `#include <R.h>`

There are many useful functions in R that can be called from your C code. You need to insert `#include <R.h>` in your code and tell your compiler where to find this file. Normally this file is located in the directory C:\Program Files\R\R-2.5.0\include. In addition to that, you need to link your code with functionality that is in R.dll. The way to do this depends on your compiler. In Microsoft Visual C proceed as follows:

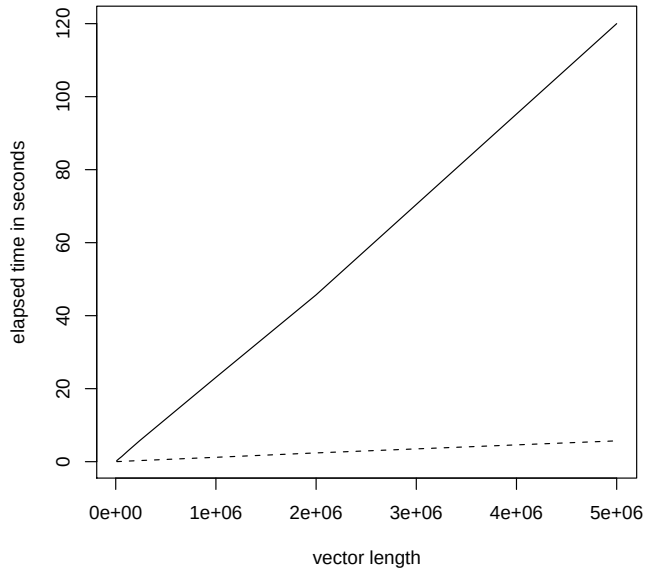

Figure 6.2: Calculation times of `arsimR` (solid line) and `arsimC` (dashed line) for increasing vectors

- Open a dos box and go to the bin directory of the R installation
- `cd` C:\Program Files\R\R-2.5.0\bin (modify as required)
- `pexports R.dll > R.exp`
- `lib /def:R.exp /out:Rdll.lib`

Here `lib` is the library command that comes with Visual C++. You can download the free Visual C++ express edition from the Microsoft site. The `pexports` tool is part of the MinGW-utils package. Now the file Rdll.lib is created and when you create your dll the compiler needs to link this lib file as well. See [5] and the R manual 'Writing R extensions' for more information. The type of R functions that can be called

- printing and error handling
- numerical and mathematical
- memory allocation

As an example we slightly modify the above arsim.c file

```
#include <R.h>

void arsim(double *x, long *n, double *phi)
```

```
{
  long i;
  Rprintf("Before the loop \n");

  if( *n > 100 )
    MESSAGE "vector is larger than 100" WARN

  for (i=1; i<*n; i++)
    x[i] = *phi * x[i-1] + x[i] ;
  Rprintf("After the loop \n");
}
```

Note that if you have loaded the dll with `dyn.load`, you must not forget to unload it with the function `dyn.unload` if you want to build a newer version. R has locked the dll and the compiler is not able to build a new version. After a successful build we can run `arsimC` again, which now gives some extra output.

```
out2 <- arsimC(rnorm(500), phi = 0.75)
Before the loop
After the loop
Warning message:
vector is larger than 100
```

6.4.3 Evaluating R expressions in C

A handy thing to do in C is evaluating R expressions or R functions. This enables you for example, to write a numerical optimization routine in C and pass an R function to that routine. Within the C routine, calls to the R function can then be made, this is just the way, or example, the R function `optim` works.

To evaluate R expressions or R functions in C, it is better to use the `.Call` or `.External` interfaces, the `eval` function is the function in C that can be used to evaluate R expressions.

```
SEXP eval(SEXP expr, SEXP rho);
```

which is the equivalent of the interpreted R code `eval(expr, envir = rho)`. See section 5.9 of the R manual 'Writing R Extensions'. The internal R pointer type `SEXP` is used to pass functions, expressions, environments and other language elements from R to C. It is defined in the file 'Rinternals.h'.

A small example

We will give a small example first that does almost nothing, but shows some important concepts. The example takes an R function and evaluates this R function in C with input argument `xinput`. First the necessary C code:

```
#include <R.h>
#include <Rinternals.h>

SEXP EvalRExpr( SEXP fn, SEXP xinput, SEXP rho)
{
  SEXP ans, R_fcall;
  int n = length(xinput);

  PROTECT(R_fcall = lang2(fn, R_NilValue));
  PROTECT(ans = allocVector(VECSXP, n));
  SETCADR(R_fcall, xinput);

  ans = eval(R_fcall, rho);
  Rprintf("Length of xinput %d \n", n);
  UNPROTECT(2);
  return ans;
}
```

When this is build into a dll that exports the function `EvalRExpr`, then we can load the dll in R and use `.Call` to run the function:

```
z <- c(121, 144, 225)
myf <- function(x)
{
  2* sqrt(x)
}

.Call("EvalRExpr",
  myf,
  as.double(z),
  new.env()
)

Length of xinput 3
[1] 22 24 30
```

First, in the C code the R objects `ans` and `R_fcall` of type `SEXP` are defined. To protect the `ans` object from the R garbage collection mechanism it is created with the `PROTECT`

macro. Enough memory is allocated with `allocVector(VECSXP, n)`, in our example we will return a vector of the same length as the input vector.

The `R_fcall` object is created with the function `lang2`, which creates an executable pair list and together with a call to `SETCADR` we can then call the function `eval` which will evaluate our R function `fn`. A call to `PROTECT` must always be accompanied with a call to `UNPROTECT`, in this example we had two calls to `PROTECT` so we call `UNPROTECT(2)` before we exit the C code.

A numerical integration example

In R the function `integrate` can calculate the integral

$$\int_a^b f(x)dx$$

for a one dimensional function f, using a numerical integration algorithm.

```
integrate(sin,0,pi)
2 with absolute error < 2.2e-14
```

As an illustration we create our own version using existing C code. Our version will also take a function name and the values a and b as input arguments. The following steps are done:

Adding the interface to R function The C code (from numerical recipes) implements the Romberg adaptive method, it consists of four functions:

- The function `qromb`, implements the Romberg method.
- The functions `polint` and `trapzd`, these are auxiliary functions used by `qromb`.
- The function `func`, this is the function that is going to be integrated.

In addition to these four C functions we add a C function `Integrate` that will act as the interface to R. The dll that we will create exports this function.

```
SEXP Integrate( SEXP fn, SEXP a, SEXP b, SEXP rho)
{
  SEXP ans;
  double mys;
  mys = qromb(REAL(a)[0], REAL(b)[0], fn, rho);

  PROTECT(ans = allocVector(REALSXP, 1));
```

```
  REAL(ans)[0] = mys;
  UNPROTECT(1);
  return ans;
}
```

The lower bound `a` and the upperbound `b` are of type `SEXP` and are passed from R to the C code, they are converted to `double` and passed to the `qromb` function. This function returns the result in the double variable `mys`, which we transform to a variable of type `SEXP` so that it can be passed to R.

The only modification to the existing C code `qromb`, is the addition of two input parameters `fn` and `rho` which will be needed when we want to evaluate the R function that is given by `fn`. In fact, the function `qromb` calls `polint` and `trapzd` that will call the function `fn`, so these functions also need to be given `fn` and `rho`.

Modifying the function `func` Normally, when you want to use the function `qromb` in a stand alone C program then the function to integrate is programmed in the function `func`. Now this function needs to be adjusted in such a way that it evaluates the function `fn` that you have given from R.

```
double func(const double x, SEXP fn, SEXP rho)
{
  SEXP R_fcall, fn_out, x_input;

  PROTECT(R_fcall = lang2(fn, R_NilValue));
  PROTECT(x_input = allocVector(REALSXP, 1));
  PROTECT(fn_out = allocVector(VECSXP, 1));

  REAL(x_input)[0] = x;
  SETCADR(R_fcall, x_input);
  fn_out = eval(R_fcall,rho);

  UNPROTECT(3);
  return REAL(fn_out)[0];
}
```

The same constructions are used as in the previous example. Evaluating the R functions results in a variable of type `SEXP`, this is then converted to a double and returned by `func`. When the dll is compiled we can link it to R and run the function.

```
mydll = "C:\\Test\\Release\\Integrate.dll"
dyn.load(mydll)
myf <- function(x)
{
```

```
  x*sin(x)
}

.Call("Integrate",
  myf,
  as.double(0),
  as.double(2),
  new.env()
)
[1] 1.741591
```

Ofcourse you could have used the R function `integrate`, as a comparison:

```
integrate(myf, 0, 2)
1.741591 with absolute error < 1.9e-14
```

it gives the same result!

7 Graphics

7.1 Introduction

One of the strengths of R above SAS or SPSS is its graphical system, there are numerous functions. You can create 'standard' graphs, use the R syntax to modify existing graphs or create completely new graphs. A good overview of the different aspects of creating graphs in R can be found in [6]. In this chapter we will first discuss the graphical functions that can be found in the base R system and the lattice package. There are more R packages that contain graphical functions, one very nice package is ggplot2, http://had.co.nz/ggplot2. We will give some examples of ggplot2 in the last section of this chapter.

The graphical functions in the base R system, can be divided into two groups:

High level plot functions These functions produce 'complete' graphics and will erase existing plots if not specified otherwise.

Low level plot functions These functions are used to add graphical objects like lines, points and texts to existing plots.

The most elementary plot function is `plot`. In its simplest form it creates a scatterplot of two input vectors.

```
x <- rnorm(50)
y <- rnorm(50)
plot(x,y)
```

To add titles to the existing plot use the low-level function `title`.

```
title("Figure 1")
```

Use the option `type="l"` (l as in line) in the `plot` function to connect consecutive points. This option is useful to plot mathematical functions. For example:

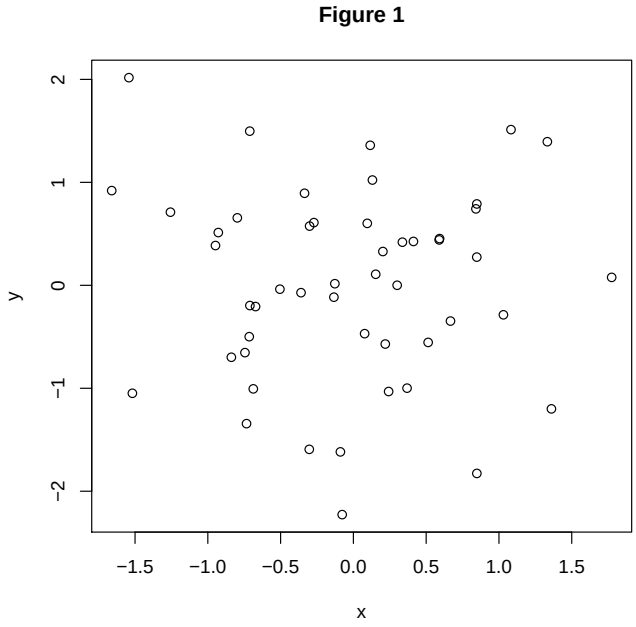

Figure 7.1: A scatterplot with a title

```
u <- seq(0, 4*pi, by=0.05)
v <- sin(u)
plot(u,v, type="l", xlab="x axis", ylab="sin")
title("figure 2")
```

In case of drawing functions or expressions, the function `curve` can be handy, it takes some work away, the above code can be replaced by the following call to produce the same graph.

```
curve(sin(x), 0,4*pi)
```

7.2 More plot functions

In this section we just mention some useful plot functions, we refer to the help files of the corresponding functions for detailed information.

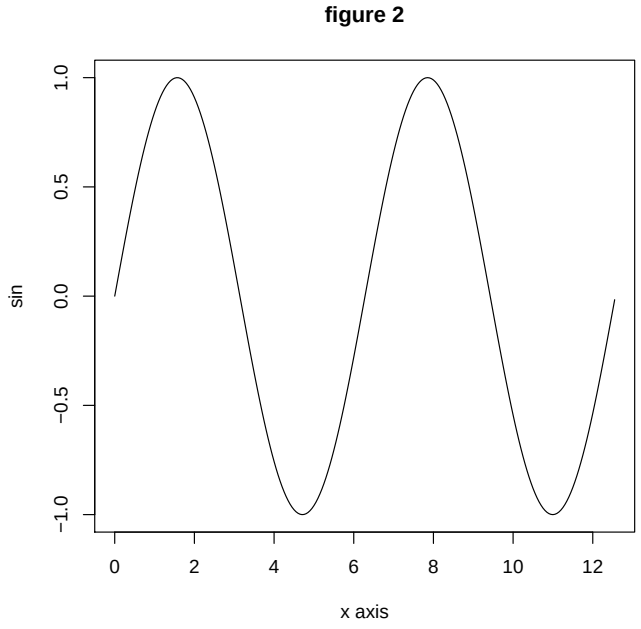

Figure 7.2: Line plot with title, can be created with `type="l"` or the `curve` function.

7.2.1 The `plot` function

The `plot` function is very versatile function. As we will see in in section 9.1 about object oriented programming, the `plot` function is a so called a generic function. Depending on the class of the input object the function will call a specific plot method. Some examples:

- `plot(xf)`, creates a bar plot if `xf` is a vector of data type factor.
- `plot(xf, y)`, creates box-and-whisker plots of the numeric data in `y` for each level of `xf`.
- `plot(x.df)`, all columns of the data frame `x.df` are plotted against each other.
- `plot(myts)`, creates a time series plot if `myts` is a ts (time series) object.
- `plot(xdate, yval)`, if `xdate` is a 'Date' object R will plot `yval` with a 'suitable' x-axis.
- `plot(xpos, y)`, creates a scatterplot where `xpos` is a POSIXct object and `y` is a numeric vector.
- `plot(f, low, up)`, creates a graph of the function `f` between `low` and `up`.

The code below shows some examples of the different uses of the function `plot`.

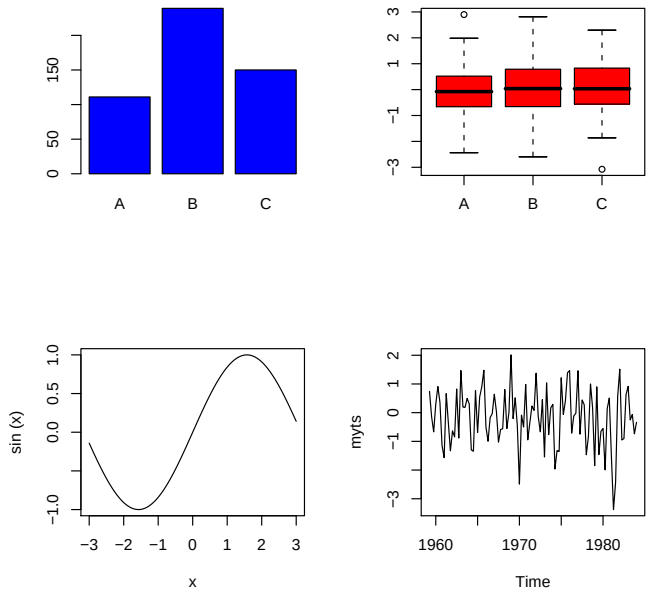

Figure 7.3: Different uses of the function `plot`

```
## set a 2 by 2 layout
par(mfrow=c(2,2))
plot(xf, col="blue")
plot(xf,rnorm(500), col="red")
plot(sin, -3,3)
plot(myts)
```

7.2.2 Distribution plots

R has a variety of plot functions to display the distribution of a data vector. Suppose the vector `x` is numeric data vector, for example:

```
x <- rnorm(1000)
```

Then the following function calls can be used to analyze the distribution of `x` graphically:

- `hist(x)`, creates a histogram.
- `qqnorm(x)`, creates a quantile-quantile plot with normal quantiles on the x-axis.
- `qqplot(x,y)`, creates a qq-plot of `x` against `y`.

- `boxplot(x)`, creates a box-and-whisker plot of `x`.

The above functions can take more arguments for fine tuning the graph, see the corresponding help files. The code below creates an example of each graph in the above list

```
x <- rnorm(100)
y <- rt(100,df=3)
par(mfrow=c(2,2))
hist(x, col=2)
qqnorm(x)
qqplot(x, y)
boxplot(x, col="green")
```

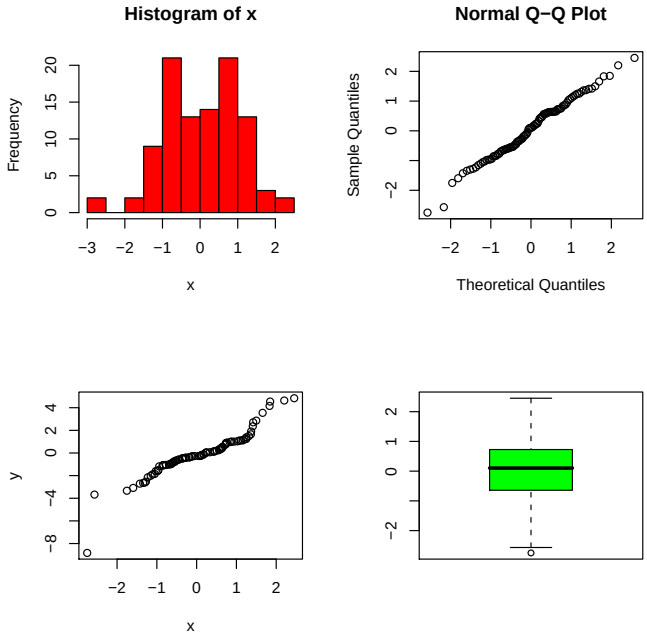

Figure 7.4: Example distribution plot in R

If you have a factor variable `x`, you can use the functions `pie` or `barplot` in combination with the `table` function to get a graphical display of the distribution of the levels of `x`. Lets look at the 'cars' data it has the factor columns 'Country' and 'Type'.

```
pie(table(cars$Country))
barplot(table(cars$Type))
```

The first argument of barplot can also be a matrix, in that case either stacked or grouped bar plots are created. This will depend on the logical argument `beside`.

```
barplot(
   table(
      cars$Country,
      cars$Type
   ),
   beside = T,
   legend.text = T
)
```

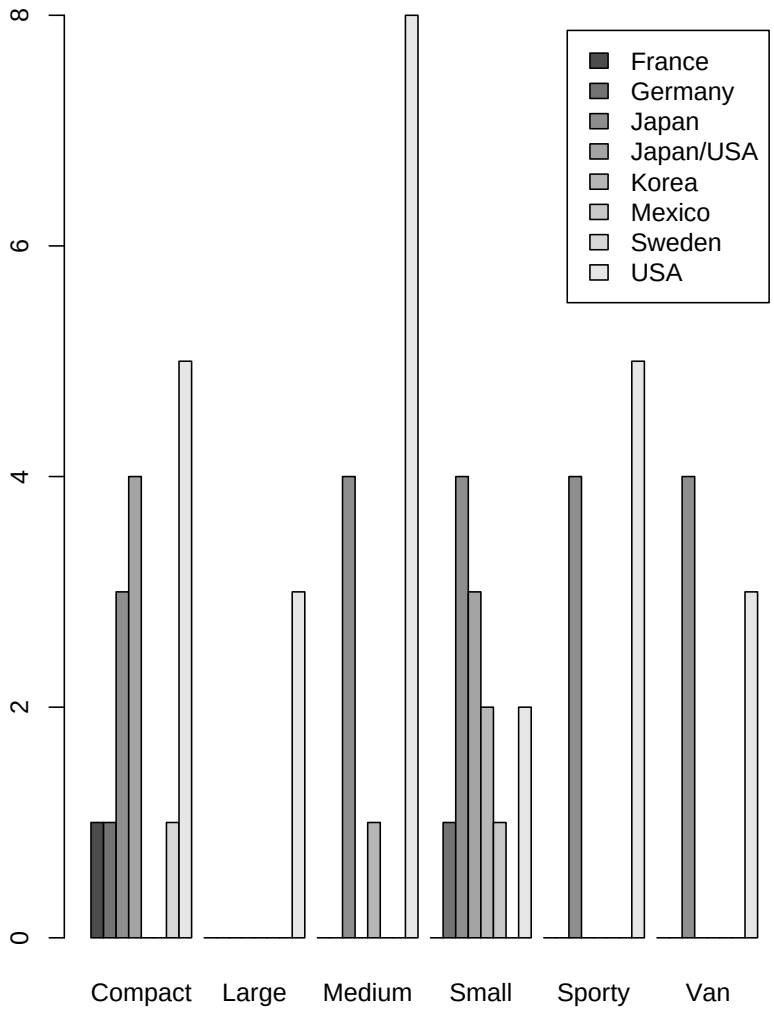

Figure 7.5: Example barplot where the first argument is a matrix

7.2.3 Two or more variables

When you have two or more variables (in a data frame) you can use the following functions to display their relationship.

- `pairs(mydf)`, when `mydf` is a data frame then each column in `mydf` is is plotted against each other, the same as plot(mydf).
- `symbols`, creates a scatterplot where the symbols can vary in size.
- `dotchart`, creates a dot plot that can be grouped by levels of a factor.
- `contour`, `image`, `filled.contour` create contour and image plots
- `persp`, creates surface plots.

In addition, multi panel graphs (Trellis graphs) described in section 7.4 can also be used to visualize multi dimensional data. The code below demonstrate some of the above functions.

```
## define some data
x <- y <- seq(-4*pi, 4*pi, len=27)
r <- sqrt(outer(x^2, y^2, "+"))
z <- cos(r^2)*exp(-r/6)

## set a 2 by 2 layout
par(mfrow=c(2,2))

image( z, axes = FALSE,
       main = "Math can be beautiful ...",
       xlab = expression(cos(r^2) * e^{-r/6})
)

dotchart( t(VADeaths),
          xlim = c(0,100),
          cex = 0.6,
          main = "Death Rates in Virginia")

## plots 'thermometers' where a proportion of the
## thermometer is filled based on Ozone value.
symbols(
  airquality$Temp,
  airquality$Wind,
  thermometers = cbind(
     0.07,
     0.3,
     airquality$Ozone / max(airquality$Ozone, na.rm=TRUE)
  ),
```

```
  inches = 0.15,
)

myf <- function(x,y)
{
  sin(x)+cos(y)
}
x <- y <- seq(0,2*pi, len=25)
z <- outer(x, y, myf)
persp(x,y,z, theta=45, phi=45, shade=0.2)
```

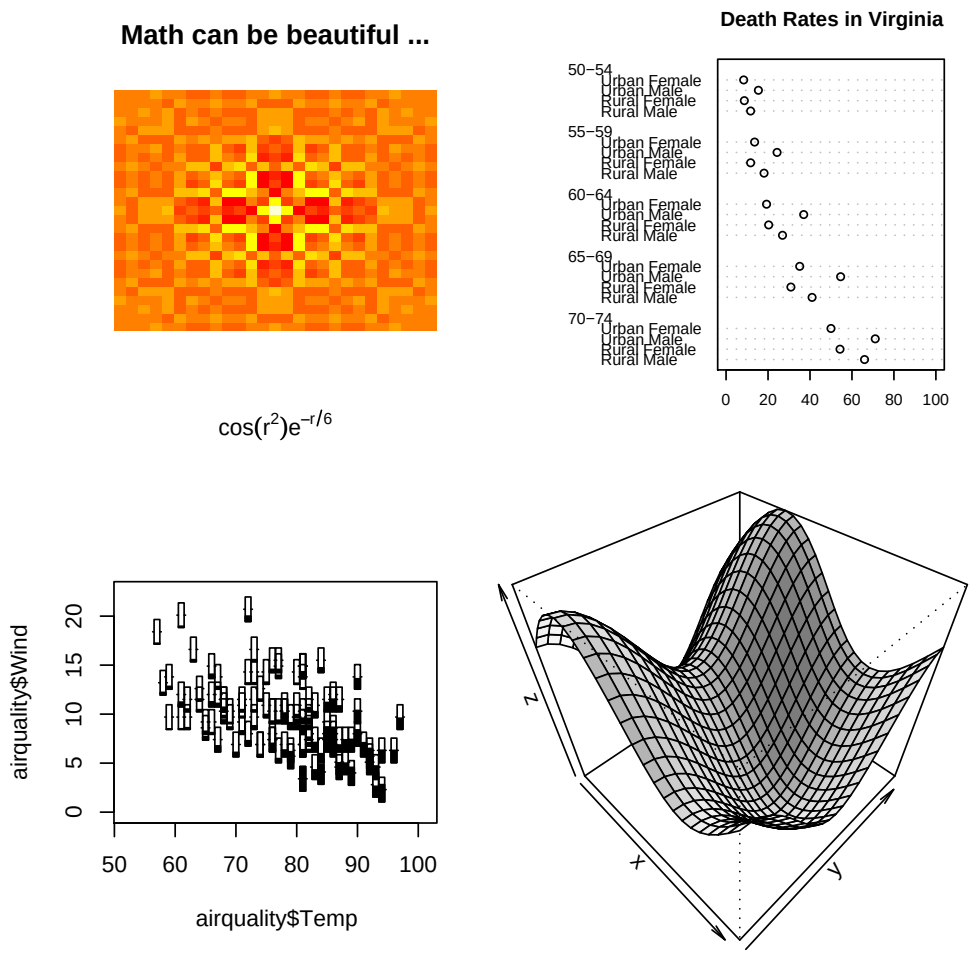

Figure 7.6: Example graphs of multi dimensional data sets

7.2.4 Graphical Devices

Before a graph can be made a so-called graphical device has to be opened. In most cases this will be a window on screen, but it may also be an eps, or pdf file. Type `?Devices` for an overview of all available devices. The devices in R are:

- `windows` The graphics driver for Windows (on screen, to printer and to Windows metafile).
- `postscript` Writes PostScript graphics commands to a file
- `pdf` Write PDF graphics commands to a file
- `pictex` Writes LaTeX/PicTeX graphics commands to a file
- `png` PNG bitmap device
- `jpeg` JPEG bitmap device
- `bmp` BMP bitmap device
- `xfig` Device for XFIG graphics file format
- `bitmap` bitmap pseudo-device via GhostScript (if available).

When a plot command is given without opening a graphical device first, then a default device is opened. Use the command `options("devices")` to see what the default device is, usually it is the windows device.

We could, however, also open a device ourselves first. The advantages of this are

- We can open the device without using the default values.
- When running several high level plot commands without explicitly opening a device only the last command will result in a visible graph, since high level plot commands overwrite existing plots. This can be prevented by opening separate devices for separate plots.

```
windows(width=8)
plot(rnorm(100))
windows(width=9)
hist(rnorm(100))
windows(width=10)
qqnorm(rnorm(100))
```

Now three devices of different size are open. A list of all open devices can be obtained by using the function `dev.list`:

```
dev.list()
windows windows windows
      2       3       4
```

When more than one device is open, there is one active device and one or more inactive devices. To find out which device is active the function `dev.cur` can be used.

```
dev.cur()
windows
      4
```

Low-level plot commands are placed on the active device. In the above example the command `title("qqplot")` will result in a title on the qqnorm graph. Another device can be made active by using the function `dev.set`.

```
dev.set(which=2)
title("Scatterplot")
```

A device can be closed using the function `dev.off`. The active device is then closed. For example, to export an R graph to a jpeg file so that it can be used in a website, use the `jpeg` device:

```
jpeg("C:\\Test.jpg")
plot(rnorm(100))
dev.off()
```

7.3 Modifying a graph

7.3.1 Graphical parameters

To change the layout of a plot or to change a certain aspect of a plot such as the line type or symbol type, you will need to change certain graphical parameters. We have seen some in the previous section. The graphical functions in R accept graphical parameters as extra arguments. These graphical parameters are usually three or four letter abbreviations (like `col`, `cex` or `mai`). The following use of the `plot` function,

```
plot(x,y, xlim=c(-3.3))
```

will set the minimum and maximum values of the x-axis. It is also possible to use the function `par` to set graphical parameters. Some graphical parameters can only be set with this function. A call to the function `par` has the following form:

```
par(gp1 = value1, gp2 = value2)
```

In the above code the graphical parameter `gp1` is set to `value1`, graphical parameter `gp2` is set to `value2` and so on. Note that some graphical parameters are read only and cannot be changed. Run the function `par` with no arguments to get a complete listing of the graphical parameters and their current values.

```
par()
$xlog
[1] FALSE

$ylog
[1] FALSE

$adj
[1] 0.5

$ann
[1] TRUE

$ask
[1] FALSE

$bg
[1] "transparent"
...
etc.
```

We will discuss some useful graphical parameters. See the help file of `par` for a more detailed description and a list of all the graphical parameters. Once you set a graphical parameter with the `par` function, that graphical parameter will keep its value until you:

- Set the graphical parameter to another value with the `par` function.
- Close the graph. R will use the default settings when you create a new plot.

When you specify a graphical parameter as an extra parameter to a graphical function, the current value of the graphical parameter will not be changed. Some example code:

```
## define some data
x <- rnorm(10)
y <- rnorm(10)
z <- rnorm(10)

## set plotting color to red
par(col="red")
plot(x,y)

## draw extra blue points
points(x,z, col="blue")
## draw red points again
points(y,z)
```

The Plot and figure regions, the margins

A graph consists of three *regions*. A 'plot region' surrounded by a 'figure regions' that is in turn surrounded by four outer margins. The top, left, bottom and right margins. See figure 7.7. Usually the high level plot functions create points and lines in the plot region.

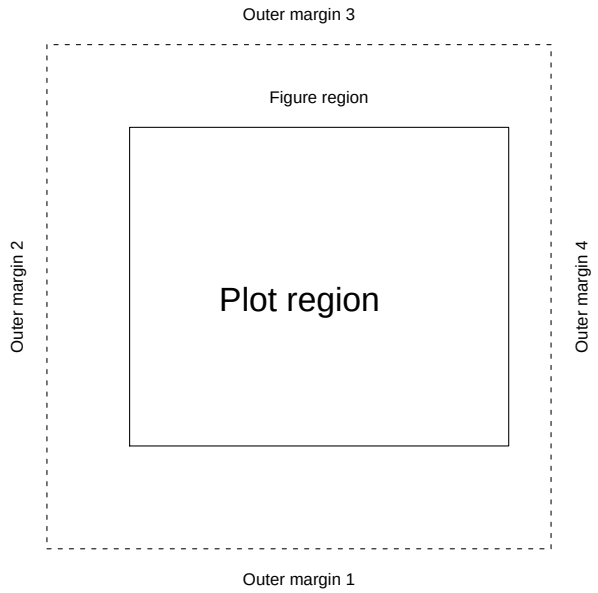

Figure 7.7: The different regions of a plot

The outer margins can be set with the `oma` parameter, the four default values are set to zero. The margins surrounding the plot region can be set with the `mar` parameter. Experiment with the `mar` and `oma` parameters to see the effects.

```
## Default values
par(c("mar", "oma"))
$mar
[1] 5.1 4.1 4.1 2.1
$oma
[1] 0 0 0 0
## set to different values
par(oma=c(1, 1, 1, 1))
par(mar=c(2.5, 2.1, 2.1, 1))
plot(rnorm(100))
```

Multiple plots on one page

Use the parameter `mfrow` or `mfcol` to create multiple graphs on one layout. Both parameters are set as follows:

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

where `r` is the number of rows and `k` the number of columns. The graphical parameter `mfrow` fills the layout by row and `mfcol` fills the layout by column. When the `mfrow` parameter is set, an empty graph window will appear and with each high-level plot command a part of the graph layout is filled. We have seen an example in the previous section, see figure 7.6.

A more flexible alternative to set the layout of a plotting window is to use the function `layout`. An example, three plots are created on one page, the first plot covers the upper half of the window. The second and third plot share the lower half of the window.

```
## first argument is a matrix with integers
## specifying the next n graphs
nf = layout(
  rbind(
    c(1,1),
    c(2,3)
  )
)
## If you are not sure how layout has divided the window
## use layout.show to display the window splits
## layout.show(nf)
plot(rnorm(100),type="l")
hist(rnorm(100))
qqnorm(runif(100))
```

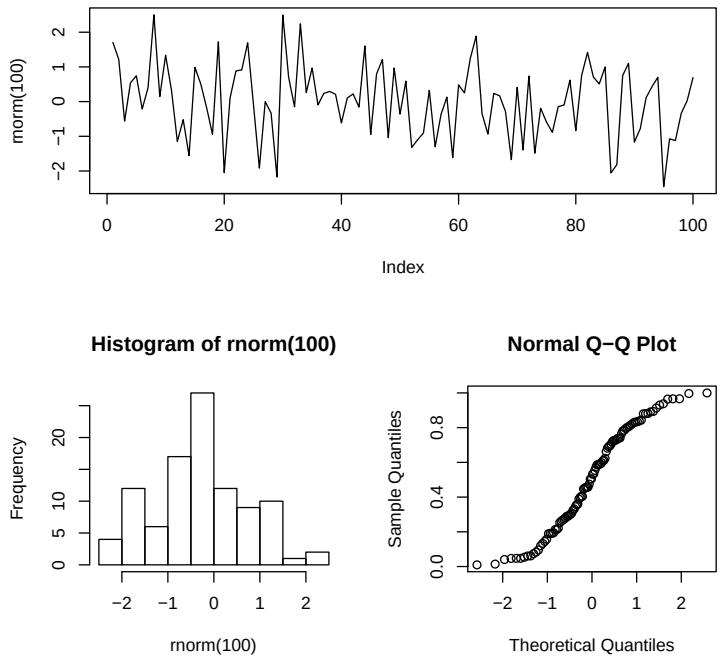

Figure 7.8: The plotting area of this graph is divided with the `layout` function.

The matrix argument in the `layout` function can contain 0's (zero's), leaving a certain sub plot empty. For example:

```
nf = layout(
  rbind(
    c(1,1),
    c(0,2)
  )
)
```

Other settings

The following list shows some more parameters, these are usually set as an argument of the plotting routine. For example, `plot(x,y, col=2)`.

- `lwd`, the line width of lines in a plot, a positive number default `lwd=1`.
- `lty`, the line type of lines in a plot, this can be a number or a character. For example `lty="dashed"`.

- `col`, the color of the plot, this can be a number or a character. For example `col = "red"`.
- `font`, an integer specifying which font to use for text on plots.
- `pch`, an integer or character that specifies the plotting symbols, in scatterplots for example.
- `xlab`, `ylab`, character strings that specify the labels of the x and y axis. Usually given direct with the high-level plotting functions like `plot` or `hist`.
- `cex`, character expansion. A numerical value that gives the amount to scale the plotting symbols and texts. The default is 1.

Some of the graphical parameters may be set as vectors so that each point, text or symbol could have its own graphical parameter. This is another way to display an additional dimension. Lets look at a plot with different symbols, for the cars data set we can plot the 'Price' and 'Mileage' variables in a scatterplot and have different symbols for the different 'Types' of cars.

```
Ncars = dim(cars)[1]
plot(
   cars$Price, cars$Mileage,
   pch = as.integer(cars$Type)
)
legend(20000,37,
    legend = levels(cars$Type),
    cex=1.25, pch=1:6
)
```

The color palette

The graphical parameter `col` can be a vector. This can be used to create a scatterplot of 'Price' and 'Mileage' where each point has a different color depending on the 'Weight' value of the car. To do this, we first need to change the color palette in R. The color palette specifies which colors corresponds with the numbers 1,2,3,... in the specification `col = number`. The current palette can be printed with the function `palette`.

```
palette()
[1] "black"   "red"     "green3"  "blue"    "cyan"
[6] "magenta" "yellow"  "gray"
```

This means `plot(rnorm(100), col=2)` will create a scatterplot with red points. The function can also be used to change the palette. Together with a few auxiliary functions (`heat.colors`, `terrain.colors`, `gray`), it is easy to create a palette of colors, say from dark to light red.

```
palette(heat.colors(Ncars))
palette()
[1] "red"     "#FF2400" "#FF4900" "#FF6D00" "#FF9200" "#FFB600"
[7] "#FFDB00" "yellow"  "#FFFF40" "#FFFFBF" ...
```

So in the color palette, `col=1` represents red, `col=2` a slightly lighter red and so on. Then in the plot function we specify `col=order(cars$Weight)`, the largest value has order number `Ncars`. The following code uses several (plot) functions to create a colored scatterplot and a color legend.

```
## split the screen in two, the larger left part will contain
## the scatter plot the right side contains a color legend
layout(matrix(c(1, 2), nc = 2), widths = c(4, 1))
## create the scatterplot with different colors
plot(
   cars$Price, cars$Mileage,
   pch = 16, cex = 1.5,
   col = order(cars$Weight)
)

## do some calculations for the color legend, determine
## minimum and maximum weight values.
zlim = range(cars$Weight, finite = TRUE)
## lets use 20 color values in the color legend
levels = pretty(zlim, 20)
## start the second plot that is the color legend
plot.new()
plot.window(
    xlim = c(0, 1),
    ylim = range(levels),
    xaxs = "i", yaxs = "i"
)
## use the function rect to draw multiple colored rectangles
rect(
    0, levels[-length(levels)],
    1, levels[-1],
    col = terrain.colors(length(levels) -  1)
)
## draw an axis on the right-hand side of the legend
axis(4)
```

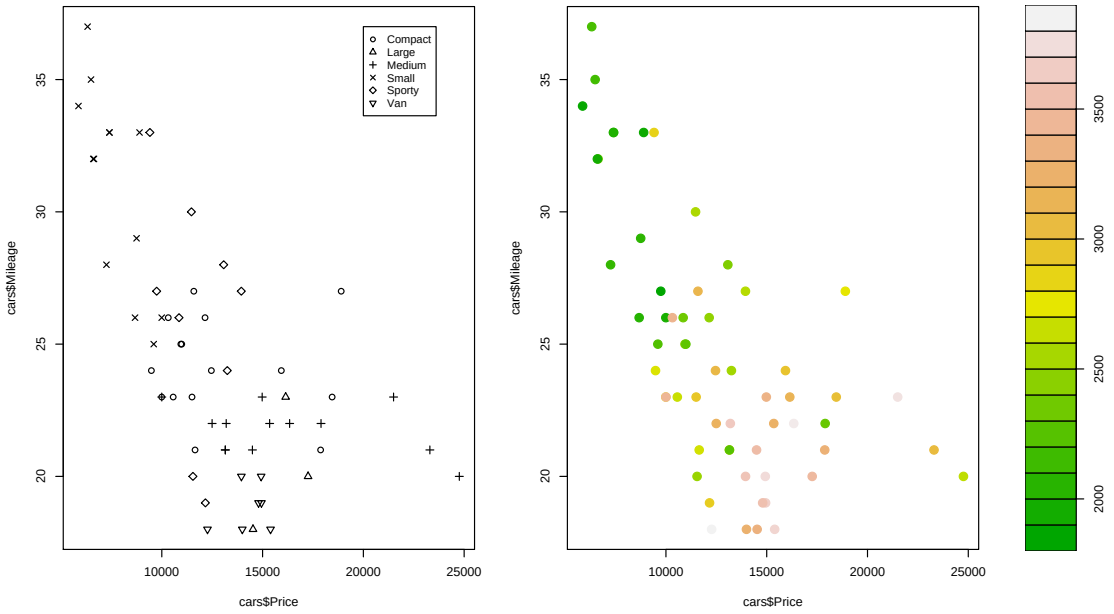

Figure 7.9: Examples of different symbols and colors in plots

To set the color palette back to default use `palette("default")`.

7.3.2 Some handy low-level functions

Once you have created a plot you may want to add something to it. This can be done with low-level plot functions.

Adding lines

The function `lines` and `abline` are used to add lines on an existing plot. The function `lines` connects points given by the input vector. The function `abline` draws straight lines with a certain slope and intercept.

```
plot(c(-2,2),c(-2,2))
lines(c(0,2), c(0,2), col="red")
abline(a=1,b=2, lty=2)
abline(v=1, lty=3, col="blue", lwd=3)
```

The functions `arrows` and `segments` are used to draw arrows and line segments.

```
## three arrows starting from the same point
## but all pointing to a different direction
arrows(
    c(0,0,0),
    c(1,1,1),
    c(0,0.5,1),
    c(1.2,1.5,1.7),
    length = 0.1
)
```

Adding points and symbols

The function `points` is used to add extra points and symbols to an existing graph. The following code adds some extra points to the previous graph.

```
points(rnorm(4), rnorm(4), pch=3, col="blue")
points(rnorm(4), rnorm(4), pch=4, cex=3, lwd=2)
points(rnorm(4), rnorm(4), pch="K", col="green")
```

Adding titles and text

The functions `title`, `legend`, `mtext` and `text` can be used to add text to an existing plot.

```
title(main="My title", sub="My subtitle")
text(0, 0, "some text")
text(1, 1, "Business & Decision", srt=45)
```

The first two arguments of `text` can be vectors specifying x,y coordinates, then the third argument must also be a vector. This character vector must have the same length and contains the texts that will be printed at the coordinates. The function `mtext` is used to place text in one of the four margins of the plot.

```
mtext("Text in the margin", side=4)
```

In R you can place ordinary text on plots, but also special symbols, Greek characters and mathematical formulas on the graph. You must use an R expression inside the `title`, `legend`, `mtext` or `text` function. This expression is interpreted as a mathematical expression, similar to the rules in LaTex.

```
text(-1.5, -1.5,
  expression(
    paste(
      frac(1, sigma*sqrt(2*pi)),
      " ",
      plain(e)^{frac(-(x-mu)^2, 2*sigma^2)}
    )
  ),
  cex = 1.2
)
```

See for more information the help of the `plotmath` function.

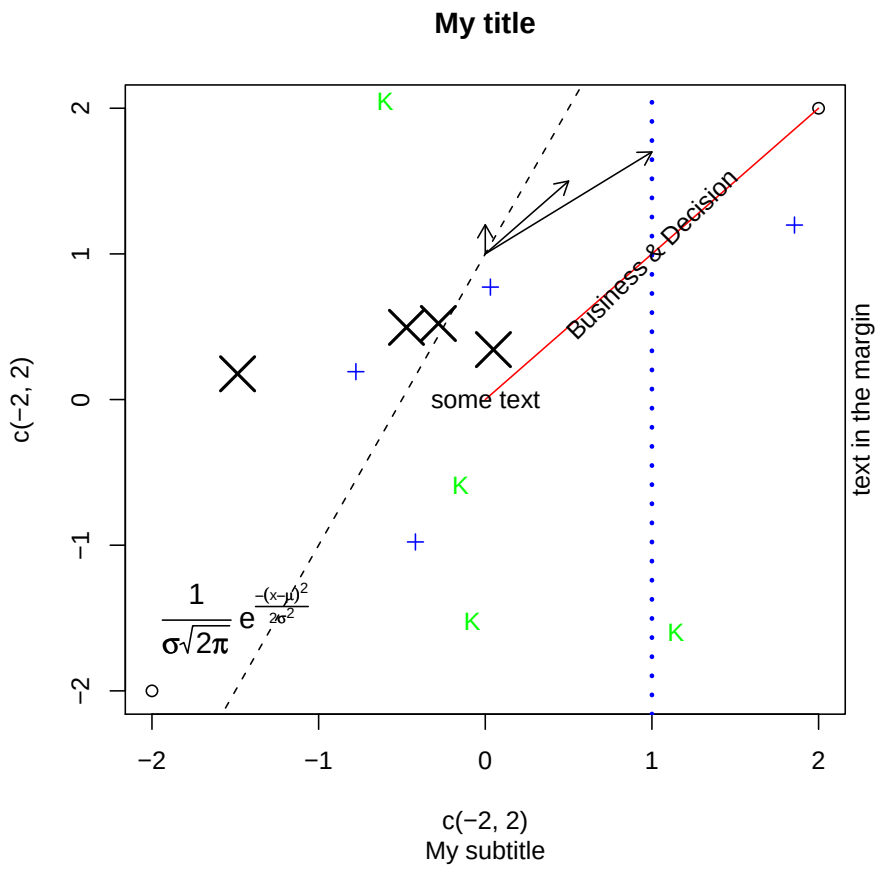

Figure 7.10: The graph that results from the previous low-level plot functions.

7.3.3 Controlling the axes

When you create a graph, the axes and the labels of the axes are drawn automatically with default settings. To change those settings you specify the graphical parameters

that control the axis, or use the `axis` function. One approach would be to first create the plot without the axis with the `axes = F` argument, and then draw the axis using the low-level `axis` function.

```
x <- rnorm(100)
y <- rnorm(100)
## do not draw the axes automatically
plot(x,y, axes=F)
## draw them manually
axis(side=1)
axis(side=2)
```

The `side` argument represents the side of the plot for the axis (1 for bottom, 2 for left, 3 for top, and 4 for right). Use the `pos` argument to specify the `x` or `y` position of the axis.

```
x <- rnorm(100)
y <- rnorm(100)
plot(x,y, axes=F)
axis(side=1, pos=0)
axis(side=2, pos=0)
```

The location of the tick marks and the labels at the tick marks can be specified with the arguments `at` and `labels` respectively.

```
## Placing tick marks at specified locations
x <- rnorm(100)
y <- rnorm(100)
plot(x,y, axes=F)
xtickplaces <- seq(-2,2,l=8)
ytickplaces <- seq(-2,2,l=6)
axis(side=1, at=xtickplaces)
axis(side=2, at=ytickplaces)

## Placing labels at the tick marks
x <- 1:20
y <- rnorm(20)
plot(x,y, axes=F)
xtickplaces <- 1:20
ytickplaces <- seq(-2,2,l=6)
xlabels <- paste("day",1:20,sep=" ")
axis(side=1, at=xtickplaces, labels=xlabels)
axis(side=2, at=ytickplaces)
```

Notice that R does not plot all the axis labels. R has a way of detecting overlap, which then prevents plotting all the labels. If you want to see all the labels you can adjust the character size, use the `cex.axis` parameter.

```
x <- 1:20
y <- rnorm(20)
plot(x,y, axes=F)
xtickplaces <- 1:20
ytickplaces <- seq(-2,2,l=6)

xlabels <- paste("day", 1:20,sep=" ")
axis(side=1, at=xtickplaces, labels=xlabels, cex.axis=0.5)
axis(side=2, at=ytickplaces)
```

Another useful parameter that you can use is `tck`. It specifies the length of tick marks as a fraction of the smaller of the width or height of the plotting region. In the extreme case`tck = 1`, grid lines are drawn.

To draw logarithmic x or y axis use `log="x"` or `log="y"`, if both axis need to be logarithmic use `log="xy"`.

```
## adding an extra axis with grid lines, this
## is on top of the existing axis.
axis(side=1,at= c(5,10,15,20), labels=rep("",5), tck=1, lty=2)

## Example of logarithmic axes
x <- runif(100,1,100000)
y <- runif(100,1,100000)
plot(x,y, log="xy", col="grey")
```

7.4 Trellis Graphics

7.4.1 Introduction

Trellis graphics add a new dimension to traditional plotting routines. They are extremely useful to visualize multi-dimensional data. You create a trellis graphic by using one of the trellis display functions, these are in the package `lattice`. Visualization of multi-dimensional data can be achieved through a multi panel layout, where each panel displays a particular subset of the data. The following table (Table 7.1) displays some of the trellis display functions in R in the lattice package.

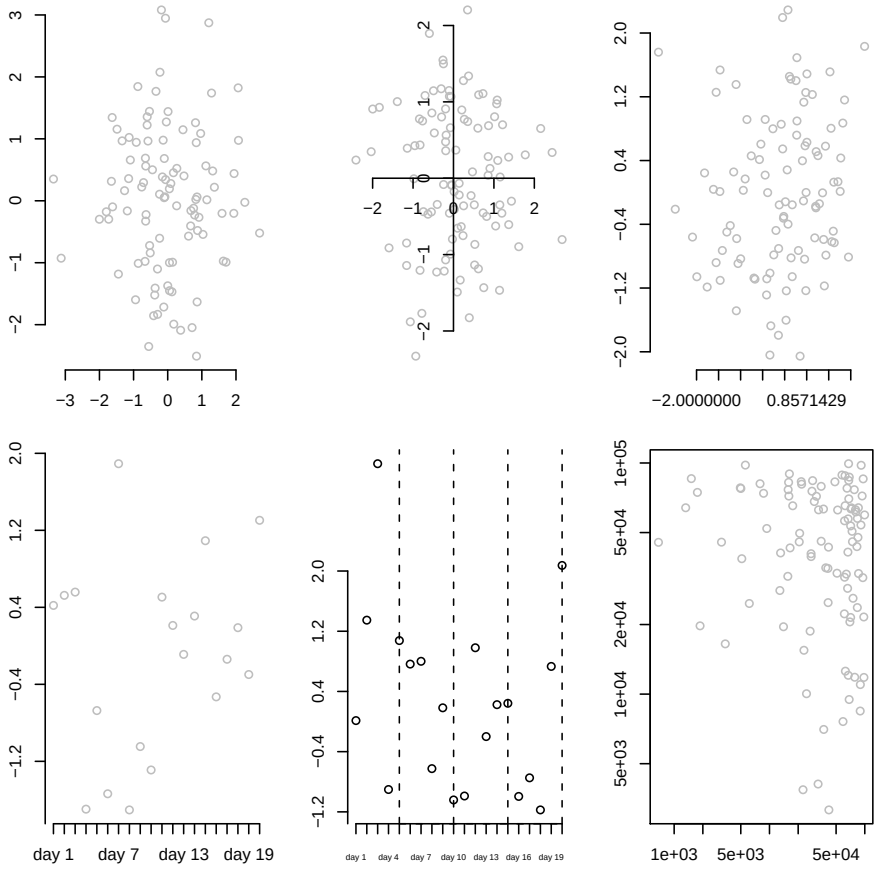

Figure 7.11: Graphs resulting from previous code examples of customizing axes.

A call to a trellis display function differs from a call to a normal plot routine. It resembles a call to one of the statistical modeling functions such as `lm` or `glm`. The call has the following form:

```
TrellisFunction(formula, data = data.frame, other graphical parameters)
```

Depending on the specific trellis display function, the formula may not have a 'response' variable. To create a scatterplot of the Price variable against the Weight variable and a histogram of the Weight variable in the car.test.frame data frame, proceed as follows:

```
cars <- read.csv("cars.csv", row.names=1)
library(lattice)
xyplot(Price ~ Weight, data=cars)
histogram( ~ Weight, data=cars)
```

Trellis function	description
barchart	Bar charts plot
bwplot	Box and whisker plot
densityplot	Kernel density plots, smoothed density estimate
dotplot	Plot of labeled data
histogram	Histogram plot
qq	Quantile-quantile plot
xyplot	Scatterplot
wireframe	3D surface plot
levelplot	Contour plot
stripplot	1 dimensional scatterplot
cloud	3D scatterplot
splom	Scatterplot matrices

Table 7.1: Trellis display functions

7.4.2 Multi panel graphs

To create a multi panel layout use the 'conditioning' operator | in the formula. To see the relationship between Price and Weight for each Type of car, use the following construction.

```
xyplot(Price ~ Weight | Type, data=cars)
```

Use the * operator to specify more than one conditioning variable. The following example demonstrates two conditioning variables. First, create some example data, one numeric variable and two grouping variables with three levels.

```
x <- rnorm(1000)
y <- sample(letters[1:3], size=1000, rep=T)
z <- sample(letters[11:13], size=1000, rep=T)
exdata <- data.frame(x,y,z)
```

Next, a histogram plot is created for the variable x conditioned on the variables y and z.

```
histogram(~x|y*z, data=exdata)
```

For each combination of the levels of y and z, a histogram plot is created. The order can be changed:

```
histogram(~x|z*y, data=exdata)
```

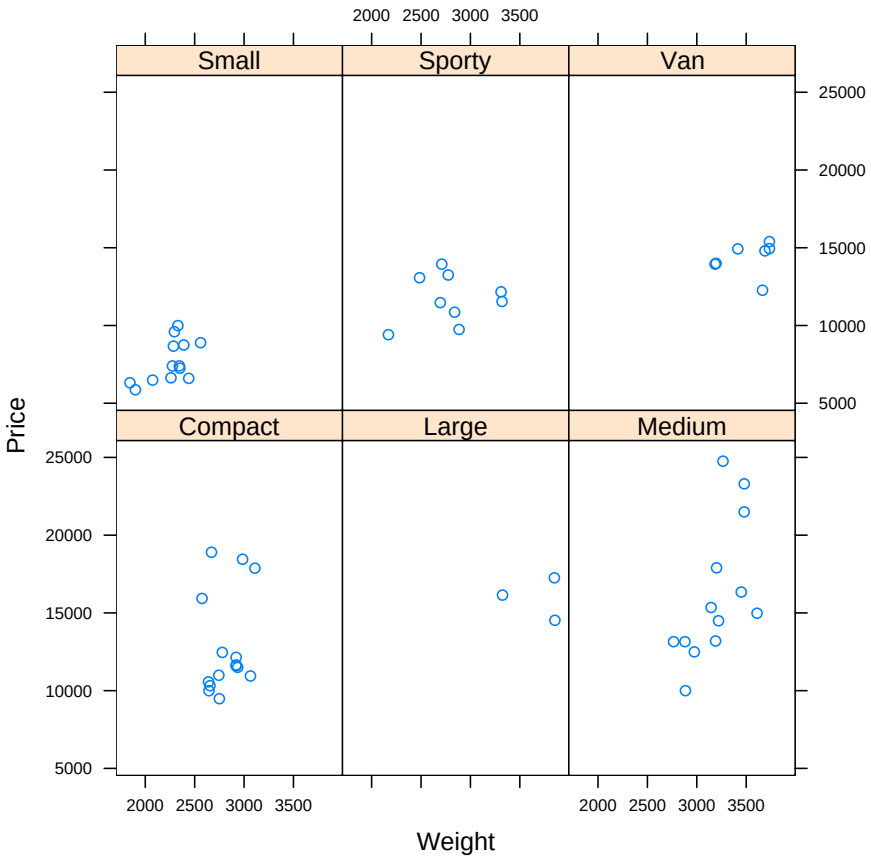

Figure 7.12: Trellis plot Price versus Weight for different types

The above examples were based on conditioning variables of type factor. In this case R will create a separate panel for each level of a factor variable or for each level combination of multiple factor variables.

To create Trellis graphics based on numeric conditioning variables you can use the functions `equal.count` or `shingle` to create conditioning intervals of numeric variables. These intervals can then be used in a Trellis display function.

Lets look at our cars example data frame that contains information on 60 different cars. Suppose we want to create a histogram of the variable 'Mileage' conditioned on the variable 'Weight'. We then proceed as follows:

```
weight.int  <- equal.count(
    cars$Weight,
    number=4, overlap=0
)
```

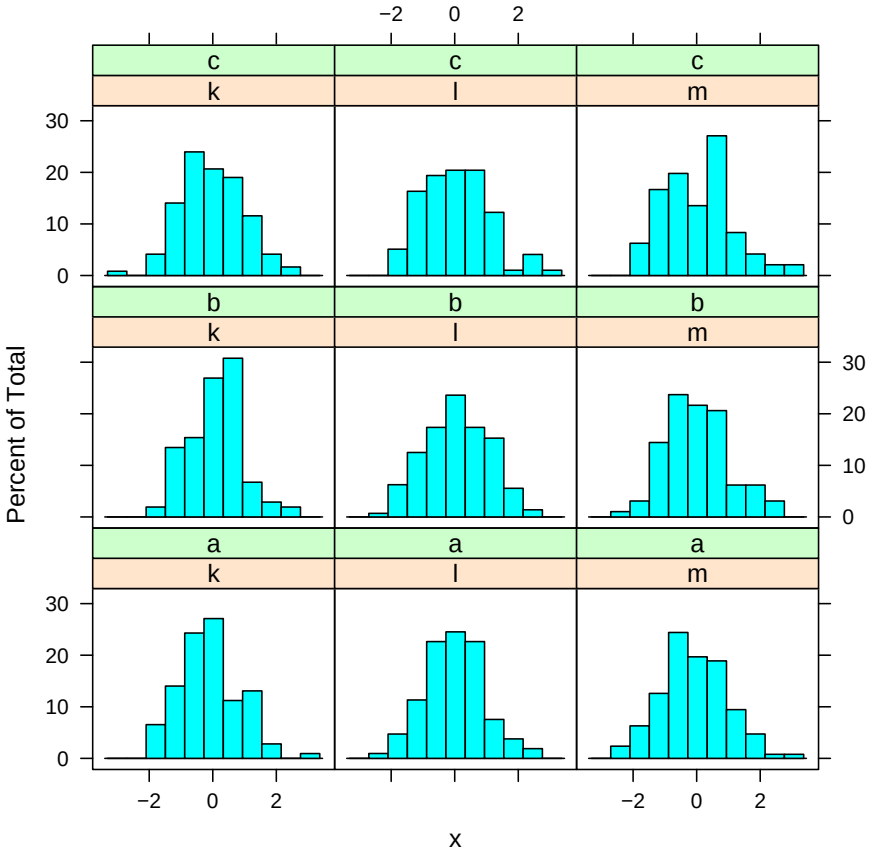

Figure 7.13: A trellis plot with two conditioning variables

This creates the conditioning intervals. The variable 'Weight' is divided into four equal intervals without overlap.

```
weight.int

Data:
 [1] 2560 2345 1845 2260 2440 2285 2275 2350 2295 1900 2390 2075 2330 3320 2885
[16] 3310 2695 2170 2710 2775 2840 2485 2670 2640 2655 3065 2750 2920 2780 2745
[31] 3110 2920 2645 2575 2935 2920 2985 3265 2880 2975 3450 3145 3190 3610 2885
[46] 3480 3200 2765 3220 3480 3325 3855 3850 3195 3735 3665 3735 3415 3185 3690

Intervals:
     min    max count
1 1842.5 2562.5    15
2 2572.5 2887.5    16
3 2882.5 3222.5    16
4 3262.5 3857.5    15
```

```
Overlap beetween adjacent intervals:
[1] 0 2 0
```

To draw the histograms use the following R code:

```
histogram( ~Mileage | weight.int ,data=cars)
```

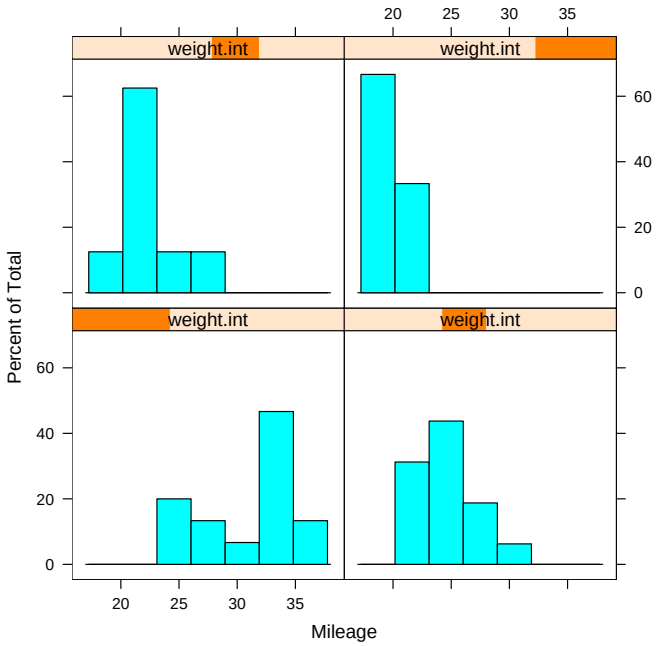

Figure 7.14: Histogram of mileage for different weight classes

7.4.3 Trellis panel functions

Trellis graphs are constructed per panel. A general trellis display function calls a panel function that does the actual work. The name of the default panel function that is called by the general trellis display function is `panel.name`, where `name` is the name of the general trellis display function. So, if we call the trellis display function `xyplot` then this will in turn call the function `panel.xyplot`. If you look at the code of the general trellis display function `xyplot` you won't see a lot. The corresponding panel function panel function `panel.xyplot` does all the work.

```
xyplot
function (x, data, ...)
```

```
UseMethod("xyplot")
<environment: namespace:lattice>

panel.xyplot
function(...)
a lot R of code
...
```

A powerful feature of trellis graphs is that you can write your own panel function and pass this function on to the general trellis display function. This is done using the argument `panel` of the trellis display function.

Suppose we want to plot 'Price' against 'Mileage' conditioned on the 'Type' variable and suppose that, in addition, we want a separate symbol for the highest price. We create our own panel function:

```
panel.maxsymbool <- function(x,y){
  biggest <- y == max(y)
  panel.points(x[!biggest],y[!biggest])
  panel.points(x[biggest], y[biggest],pch="M")
}
```

The above function first finds out what the maximum `y` value is, it then plots the points without the maximum `y` value and then plots the maximum `y` value using a different symbol. Note that we use the function `panel.points` instead of the normal low-level `points` function.

The normal low-level functions can not be used inside a function that is going to be used as panel function. This is because lattice panel functions need to use grid graphics. So use the panel versions: `panel.points`, `panel.text`, `panel.abline`, `panel.lines` and `panel.segments`.

Once a panel function is defined you should pass it to the trellis display function

```
xyplot( Price ~ Mileage | Type, data=cars, panel = panel.maxsymbol)
```

The following example fits a least squares line through the points of each panel. Additional graphical parameters can also be passed on. The next example enables the user to specify the type of line using the graphical parameter lty.

```
panel.lsline <- function(x,y, ...){
  coef <- lsfit(x,y)$coef
  panel.points(x,y)
  panel.abline(coef[1],coef[2], ...)
}
xyplot(Price~Mileage|Type,data=cars, panel=panel.lsline, lty=2)
```

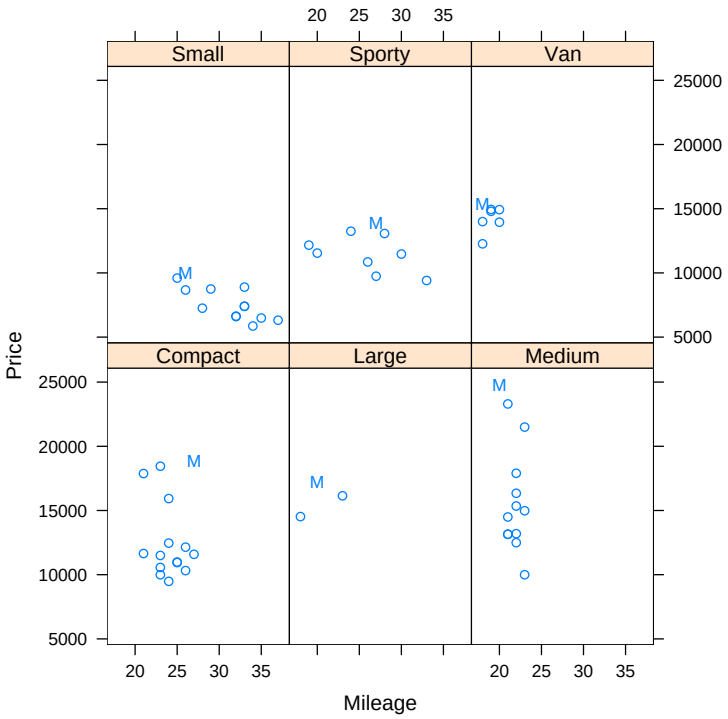

Figure 7.15: Trellis plot with modified panel function

7.4.4 Conditioning plots

The function `coplot` can be a nice alternative to the trellis function `xyplot`. It can also create multi panel layouts where each panel represents a part of the data. The function has many arguments that can be set. A few examples are given below.

```
## no need to specify intervals, only the number of intervals
coplot(lat~long | depth, number=4, data=quakes, col="blue")
## two conditioning variables
coplot(lat~long | depth*mag, number=c(4,5), data=quakes)
## conditioning on a factor and numeric variable
coplot(Price~Mileage | Type*Weight, number=3 ,data=cars)
```

The function `coplot` can also use a customized panel function, the `points` function is used as default panel function. The following example uses the function `panel.smooth` as panel function.

```
coplot(Price ~ Mileage | Weight,
   number=4 ,
   panel = panel.smooth,
```

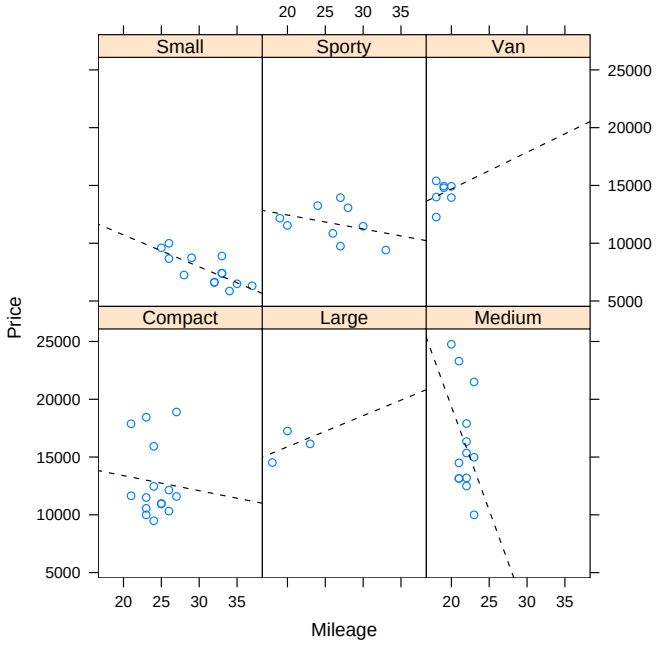

Figure 7.16: Trellis plot adding a least squares line in each panel

```
    data=cars,
    col= "dark green",
    pch=2
)
```

7.5 The ggplot2 package

The ggplot2 package (see [7]) is a collection of plotting routines based on the grammer of graphics, see [8]. The functions in ggplot2 can take away some of the anoying extra code that makes plotting a hassle (like drawing legends). At the same time ggplot2 provides a powerful model of graphics that makes it easy to produce complex multi-layered graphics. This section only gives a brief introduction, for a thorough description of the possibilities see http://had.co.nz/ggplot2.

7.5.1 The `qplot` function

The function `qplot` (quick plot) in ggplot2 can be used to create complex plots with little coding effort. The following code displays some examples.

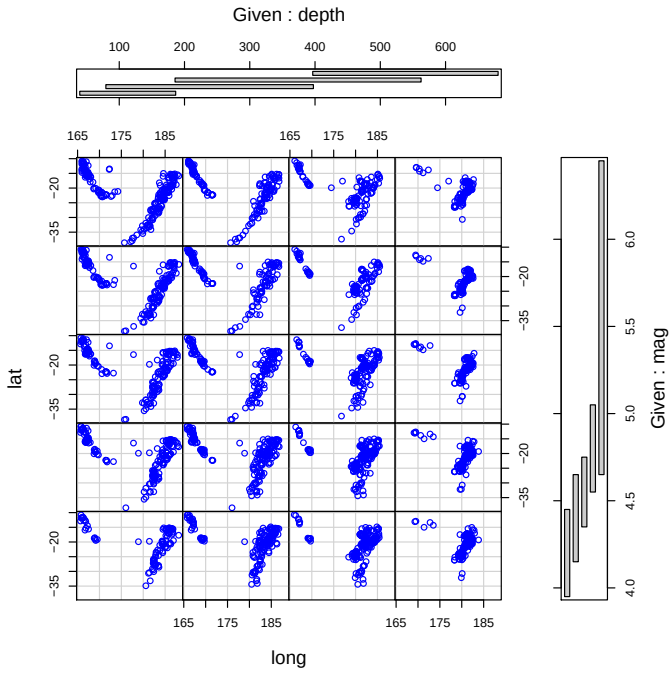

Figure 7.17: A coplot with two conditioning variables

```
library(ggplot2)
x <- runif(1000)
y <- 2*x^2 + rnorm(1000,0,0.3)
z <- 5*log(x) + rnorm(1000,0,0.3)
testdata <- data.frame(x, y, z)
## a simple scatter plot
qplot(x, y, data = testdata)
## transformations
qplot(x, log(y), data = testdata)
## changing the size of the symbols
qplot(x, y, data = testdata, size=z)
```

The function `qplot` can also create other types of plots. This can be done by using the argument `geom`, which stands for *geometric object*. Such an object not only describes the type of plot but also a corresponding statistical calculation. For example, a smoothing line calculated according to some smoothing algorithm.

The default value for `geom` is `point`, a standard scatter plot. To draw a line graph between the points use:

```
qplot(x,y, data = testdata, geom=c("line"))
```

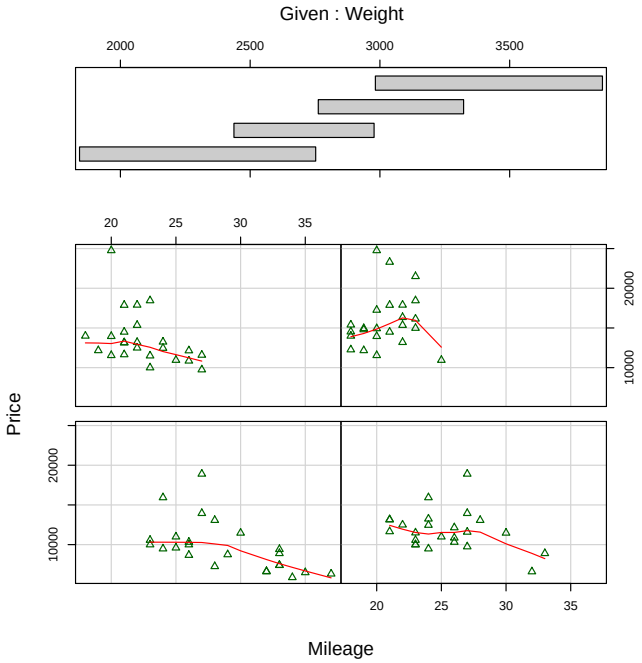

Figure 7.18: A coplot with a smoothing line

The `geom` argument can be a vector of names, this will result in one plot with multiple graphs on top of each other. The following code first plots a scatter plot with a loess smoothing line, and then a scatter plot with a regression line, using `lm`.

```
qplot(x,y, data = testdata, geom=c("point", "smooth"), span = 0.2)
qplot(x,y, data = testdata, geom=c("point", "smooth"), method = "lm")
```

7.5.2 Facetting

Facetting in ggplot2 is the equivalent of trellis plots and allows you to display certain sub sets of your data in different *facets*.

```
x <- rnorm(1000)
G1 <- sample(c("A", "B", "C"), size=1000,rep=T)
G2 <- sample(c("X", "Y", "Z"),size=1000,rep=T)
testdata <- data.frame(x,G1,G2)
qplot(x,data = testdata, facets = G1~G2, geom="histogram")
```

7.5.3 Plots with several layers

Although the function qplot is enough for creating many plots, the use of the ggplot function in combination with geometric object functions is a much more powerful way to create plots that consists of several layers.

the function ggplot

```
c <- ggplot(data = testdata, aes(y=y,x=x) )
c + stat_smooth(colour=3, size=6) + geom_point()
```

8 Statistics

The base installation of R contains many functions for calculating statistical summaries, data analysis and statistical modeling. Even more functions are available in all the R packages on CRAN. In this section we will discuss only some of these functions. For a more comprehensive overview of the statistical possibilities see for example [9] and [10].

8.1 Basic statistical functions

8.1.1 Statistical summaries and tests

A number of functions return statistical summaries and tests. The following table contains a list of only some of the statistical functions in R. The names of the functions usually speak for themselves.

Function	**purpose**
`acf(x, plot=F)`	auto or partial correlation coefficients
`chisq.test(x)`	chi squared goodness of fit test
`cor(x,y)`	correlation coefficient
`ks.test(z)`	Kolmogorov-Smirnov goodness of fit test
`mad(x)`	median absolute deviation
`mean(x)`	mean
`mean(x, trim=a)`	trimmed mean
`median(x)`	median
`quantile(x, probs)`	sample quantile at given probabilities
`range(x)`	the range, i.e. the vector c(min(x), max(x))
`stem(x)`	stem-and-leaf-plot
`t.test(x,...)`	One or two sample Student's t-test
`var(x)`	variance of x or covariance matrix of x
`var(x,y)`	covariance
`var.test(x,y)`	test on variance equality of x and y

Table 8.1: Some functions that calculate statistical summaries.

The remainder of this sub section will give some examples of the above functions.

quantiles

The `quantile` function needs two vectors as input. The first one contains the observations, the second one contains the probabilities corresponding to the quantiles. The function returns the empirical quantiles of the first data vector. To calculate the 5 and 10 percent quantile of a sample from a N(0,1) distribution, proceed as follows:

```
x <- rnorm(100)
xq <- quantile(x,c(0.05,0.1))
xq
       5%       10%
-1.496649 -1.205602
```

The function returns a vector with the quantiles as named elements.

stem-and-leaf-plots

A stem-and-leaf-plot of `x` is generated by:

```
stem(x)

N = 100   Median = -0.014053
Quartiles = -0.676618, 0.749655

Decimal point is at the colon

 The decimal point is at the |

  -3 | 5
  -2 | 721
  -1 | 654422222111000000
  -0 | 9888776665555444444333333222111100
   0 | 11123334556666777778888889
   1 | 000012224444788
   2 | 01
   3 | 3
```

distribution tests

To test if a data vector is drawn from a certain distribution the function `ks.test` can be used.

```
x <- runif(100)
out = ks.test(x,"pnorm")
out
One-sample Kolmogorov-Smirnov test

data:  x
D = 0.5003, p-value < 2.2e-16
alternative hypothesis: two-sided
```

The output object `out` is an object of class 'htest'. It is a list with five components.

```
names(out)
[1] "statistic"   "p.value"     "alternative" "method"      "data.name"
out$statistic
        D
0.5003282
```

The function can also be used to test if two data vectors are drawn from the same distribution.

```
x1 = rnorm(100)
x2 = rnorm(100)
ks.test(x1,x2)
Two-sample Kolmogorov-Smirnov test

data:  x1 and x2
D = 0.1, p-value = 0.6994
alternative hypothesis: two-sided
```

Alternative functions that can be used are `chisq.test`, `shapiro.test` and `wilcox.test`.

Note that the functions in table 8.1 usually require a vector with data as input. To calculate for example the median value of a column in a data frame: Either access the column directly or use the function `with`.

```
median (cars$Price)
[1] 12215.5

with(
  cars,
  mean(Price)
)
```

Some functions accept a matrix as input. For example, the mean of a matrix `x`, `mean(x)`, will calculate the mean of all elements in the matrix `x`. The function `var` applied on a matrix `x` will calculate the covariances between the columns of the matrix `x`.

```
x <- matrix(rnorm(99),ncol=3)
var(x)
           [,1]       [,2]       [,3]
[1,]  1.4029791 -0.1047594  0.1188696
[2,] -0.1047594  1.0752726 -0.0587097
[3,]  0.1188696 -0.0587097  0.8468122
```

The function `summary` is convenient for calculating basic statistics of columns of a data frame.

```
summary(cars)
     Price            Country    Reliability        Mileage
 Min.   : 5866   USA      :26   Min.   : 1.000   Min.   :18.00
 1st Qu.: 9932   Japan    :19   1st Qu.: 2.000   1st Qu.:21.00
 Median :12216   Japan/USA: 7   Median : 3.000   Median :23.00
 Mean   :12616   Korea    : 3   Mean   : 3.388   Mean   :24.58
 3rd Qu.:14933   Germany  : 2   3rd Qu.: 5.000   3rd Qu.:27.00
 Max.   :24760   France   : 1   Max.   : 5.000   Max.   :37.00
                 (Other)  : 2   NA's   :11.000
     Type      Weight         Disp.             HP
Compact:15  Min.   :1845   Min.   : 73.0   Min.   : 63.0
Large  : 3  1st Qu.:2571   1st Qu.:113.8   1st Qu.:101.5
Medium :13  Median :2885   Median :144.5   Median :111.5
Small  :13  Mean   :2901   Mean   :152.1   Mean   :122.3
Sporty : 9  3rd Qu.:3231   3rd Qu.:180.0   3rd Qu.:142.8
Van    : 7  Max.   :3855   Max.   :305.0   Max.   :225.0
```

8.1.2 Probability distributions and random numbers

Most of the probability distributions are implemented in R and each of the distributions has four 'flavors': the cumulative probability distribution function, the probability density function, the quantile function and the random sample generator. The names of these functions consist of the code for the distribution preceded by a letter indicating the desired flavor.

- p cumulative probability distribution function
- d probability density function
- q quantile function
- r random sample

For example, the corresponding commands for the normal distribution are:

```
pnorm(x,m,s)
dnorm(x,m,s)
qnorm(p,m,s)
rnorm(n,m,s)
```

In these expressions `m` and `s` are optional arguments representing the mean and standard deviation (not the variance!); `p` is the probability and `n` the number of random draws to be generated.

The next table gives an overview of the available distributions in R with the corresponding parameters. Don't forget to precede the code with p, d, q or r (for example pbeta or qgamma). The column 'Defaults' specifies the default values of the parameters. If there are no default values, you must specify them in the function call. For example, `rnorm(100)` will run, but `rbeta(100)` will not.

The following code generates 1000 random standard normal numbers with 5% contamination, using the `ifelse` function.

```
x <- rnorm(1000)
cont <- rnorm(1000,0,10)
p <- runif(1000)
z <- ifelse(p < 0.95,x,cont)
```

The function `sample` randomly samples from a given vector. By default it samples without replacement and by default the sample size is equal to the length of the input vector. Consequently, the following statement will produce a random permutation of the elements 1 to 50:

Code	Distribution	Parameters	Defaults
beta	beta	shape1, shape2	–, –
binom	binomial	size, prob	–, –
cauchy	Cauchy	location, scale	0, 1
chisq	chi squared	df, ncp	-, 1
exp	exponential	rate	1
f	F	df1, df2	–, –
gamma	gamma	shape, rate, scale	–, 1, 1/rate
geom	geometric	prob	–
hyper	hyper geometric	m, n, k	–, –, –
lnorm	lognormal	meanlog, sdlog	0, 1
logis	logistic	location, scale	0, 1
nbinom	negative binomial	size, prob, mu	–, –, –
norm	normal (Gaussian)	mean, sd	0, 1
pois	Poisson	Lambda	1
t	Student's t	df, ncp	–,0
unif	uniform	min, max	0, 1
weibull	Weibull	shape, scale	–, 1
wilcoxon	Wilcoxon	m, n	–, –

Table 8.2: Probability distributions in R

```
x <- 1:50
y <- sample(x)
y
 [1] 48 20 10 37 39 16 11  1 45 42 27 49 14 38 18  5 44 41  2 22
[21] 50 33 25 12 24 34 30  6 43 13 15 40 31  4 35 36 26 19 32 47
[41] 17 21  7 28  3 23 29 46  8  9
```

To randomly sample three elements from `x` use

```
sample(x,3)
[1] 13  4  1
```

To sample three elements from `x` with replacement use

```
sample(x,3,rep=T)
[1] 13 6 9
```

To randomly select five cars from the data frame 'cars' proceed as follows:

```
x <- sample(1:dim(cars)[1], 5)
cars[x,]
                      Weight Disp. Mileage     Fuel    Type
        Toyota Camry 4   2920   122      27 3.703704 Compact
       Acura Legend V6   3265   163      20 5.000000  Medium
        Ford Festiva 4   1845    81      37 2.702703   Small
         Honda Civic 4   2260    91      32 3.125000   Small
Dodge Grand Caravan V6   3735   202      18 5.555556     Van
```

There are a couple of algorithms implemented in R to generate random numbers, look at the help of the function `set.seed`, `?set.seed` to see an overview. The algorithms need initial values to generate random numbers the so-called seed of a random number generator. These initial numbers are stored in the S vector `.Random.seed`.

Every time random numbers are generated, the vector `.Random.seed` is modified, which means that the next random numbers differ from the previous ones. If you need to reproduce your numbers, you need to manually set the seed with the `set.seed` function.

```
set.seed(12)
rnorm(5)
[1] -1.258 0.710 1.807 -2.229 -1.429
rnorm(5) # different random numbers
set.seed(12)
rnorm(5) # the same numbers as the first call
[1] -1.258 0.710  1.807 -2.229 -1.429
```

8.2 Regression models

8.2.1 Formula objects

R has many routines to fit and analyse statistical models. In general, these models are used by calling a modeling function (like `lm`, `tree`, `glm`, `nls` or `coxph`) with a so-called *formula* object and additional arguments.

Formula objects play a very important role in statistical modeling in R, they are used to specify the model to be fitted. The exact meaning of a formula object depends on the modeling function. We will look at some examples in the following sections. The general form is given by:

```
response ~ expression
```

Sometimes the term `response` can be omitted, `expression` is a collection of variables combined by operators. Some examples of formula objects:

```
myform1 <- y ~ x1 + x2
myform2 <- log(y) ~ sqrt(x1) + x2:x3
myform1
y ~ x1 + x2
myform2
log(y) ~ sqrt(x1) + x2:x3
data.class(myform2)
"formula"
```

A description of formulating models using formulas is given in the various chapters of [9]. The next sections will give some examples of different statistical models in R.

8.3 Linear regression models

8.3.1 Formula objects

R can fit linear regression models of the form

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + \epsilon$$

where $\beta = (\beta_0, \cdots, \beta_p)$ are the intercept and p regression coefficients and $x_1, \cdots, x_p$ the p regression variables. The error term ϵ has mean zero and is often modeled as a normal distribution with some variance.

For two regression variables you can use the function `lm` with the following formula

```
y ~ x1 + x2
```

By default R includes the intercept of the linear regression model. To omit the intercept use the formula:

```
y ~ -1 + x1 + x2
```

Be aware of the special meaning of the operators `*`, `-`, `^`, `\` and `:` in linear model formulae. They are not used for the normal multiplication, subtraction, power and division.

The `:` operator is used to model interaction terms in linear models. The next formula includes an interaction term between the variable x_1 and the variable x_2

```
y ~ x1 + x2 + x1:x2
```

which corresponds to the linear regression model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + \epsilon$$

There is a short hand notation for the above formula which is given by

```
y ~ x1*x2
```

In general, `x1*x2*...*xp` is a short hand notation for the model that includes all single terms, order 2 interactions, order 3 interactions, ..., order p interactions. To see all the terms that are generated use the *terms* function.

```
myform <- y ~ x1*x2*x3*x4
terms(myform)
# ignoring some other output generated by terms
attr(,"term.labels")
 [1] "x1"          "x2"          "x3"          "x4"
 [5] "x1:x2"       "x1:x3"       "x2:x3"       "x1:x4"
 [9] "x2:x4"       "x3:x4"       "x1:x2:x3"    "x1:x2:x4"
[13] "x1:x3:x4"    "x2:x3:x4"    "x1:x2:x3:x4"
```

The `^` operator is used to generate interaction terms up to a certain order.

```
y ~ (x1+x2+x3)^2
```

The above formula is equivalent to

```
y ~ x1 + x2 + x3 + x1:x2 + x2:x3 + x1:x3
```

The `-` operator is used to leave out terms in a formula. We have already seen that `-1` removes the intercept in a regression formula. For example, to leave out a specific interaction term in the above model use:

```
y ~ (x1+x2+x3)^2 - x2:x3
```

which is equivalent to

```
y ~ x1 + x2 + x3 + x1:x2 + x1:x3
```

The function `I` is used to suppress the specific meaning of the operators in a linear regression model. For example, if you want to include a transformed `x2` variable in your model, say multiplied by 2, the following formula will not work:

```
y ~ x1 + 2*x2
```

The `*` operator already has a specific meaning, so you should use the following construction:

```
y ~ x1 + I(2*x2)
```

You should also use the `I` function when you want to include a 'centered' regression variable in your model. The following formula will work, however, it does not return the expected result.

```
y ~ x1 + (x2 - constant)
```

Use the following formula instead:

```
y ~ x1 + I(x2 - constant)
```

8.3.2 Modeling functions

Linear regression models are widely used to model linear relationships between different variables. There are many different functions in R to fit and analyze linear regression models. The main function for linear regression is `lm` and its main arguments are:

```
lm(formula, data, weights, subset, na.action)
```

As an example we will use our 'cars' data set to fit the following linear regression model.

$$\texttt{Weight} = \beta_0 + \beta_1 \times \texttt{Mileage} + \epsilon$$

In R this model is formulated and fitted as follows

```
cars.lm <- lm( Weight ~ Mileage , data = cars)
```

The result of the function `lm` is stored in the object 'cars.lm', which is an object of class 'lm'. To print the object, simply enter its name in the R console window.

```
cars.lm

Call:
lm(formula = Weight ~ Mileage, data = fuel.frame)

Coefficients:
 (Intercept)   Mileage
     5057.83 -87.74223

Degrees of freedom: 60 total; 58 residual
Residual standard error: 265.1798
```

Objects of class lm, and almost every other object resulting from statistical modeling functions, have their own printing method in S-PLUS. What you see when you type in `cars.lm` is not the complete content of the `cars.lm` object. Use the function `print.default` to see the complete object.

```
print.default(cars.lm)
$coefficients
(Intercept)     Mileage
 5057.82990   -87.74223

$residuals
               Eagle Summit 4                Ford Escort   4
                   397.663796                     182.663796
               Ford Festiva 4                  Honda Civic 4
                    33.632728                       9.921563
              Mazda Protege 4               Mercury Tracer 4
                   189.921563                    -491.531836
             ...
             ...
             ...
```

As you can see the object `cars.lm` is in fact a list with named components; `coefficients`, `residuals` etc. Use the function `names` to retrieve all the component names of the cars.lm objects

```
names(cars.lm)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
```

So `cars.lm` contains much more information than you would see by just printing it. The next table gives an overview of some generic functions, which can be used to extract information or to create diagnostic plots from the `cars.lm` object.

generic function	meaning
`summary(object)`	returns a summary of the fitted model
`coef(object)`	extracts the estimated model parameters
`resid(object)`	extracts the model residuals of the fitted model
`fitted(object)`	returns the fitted values of the model
`deviance(object)`	returns the residual sum of squares
`anova(object)`	returns an anova table
`predict(object)`	returns predictions
`plot(object)`	create diagnostic plots

Table 8.3: List of functions that accept an lm object

These functions are generic. They will also work on objects returned by other statistical modeling functions. The `summary` function is useful to get some extra information of the fitted model such as t-values, standard errors and correlations between parameters.

```
summary(cars.lm)

Call:
lm(formula = Weight ~ Mileage, data = cars)

Residuals:
     Min       1Q   Median       3Q      Max
-569.274 -159.073    8.793  191.494  570.241

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 5057.830    180.402   28.04   <2e-16 ***
Mileage      -87.742      7.205  -12.18   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 265.2 on 58 degrees of freedom
Multiple R-Squared: 0.7189,     Adjusted R-squared: 0.714
F-statistic: 148.3 on 1 and 58 DF,  p-value: < 2.2e-16
```

Model diagnostics

The object `cars.lm` object can be used for further analysis. For example, model diagnostics:

- Are residuals normally distributed?
- Are the relations between response and regression variables linear?
- Are there outliers?

Use the Kolmogorov-Smirnov test to check if the model residuals are normally distributed. Proceed as follows:

```
cars.residuals <- resid(cars.lm)
ks.test( cars.residuals, "pnorm",
         mean = mean(cars.residuals),
         sd = sd(cars.residuals)
)

        One-sample Kolmogorov-Smirnov test

data:  cars.residuals
D = 0.0564, p-value = 0.9854
alternative hypothesis: two-sided
```

Or draw a histogram or qqplot to get a feeling for the distribution of the residuals

```
par(mfrow=c(1,2))
hist(cars.residuals)
qqnorm(cars.residuals)
```

A plot of the residuals against the fitted value can detect if the linear relation between the response and the regression variables is sufficient. A Cooke's distance plot can detect outlying values in your data set. R can construct both plots from the `cars.lm` object.

```
par(mfrow=c(1,2))
plot(cars.lm, which=1)
plot(cars.lm, which=4)
```

Updating a linear model

Some useful functions to update (or change) linear models are given by:

`add1` This function is used to see what, in terms of sums of squares and residual sums of squares, the result is of adding extra terms (variables) to the model. The 'cars' data set also has an 'Disp.' variable representing the engine displacement.

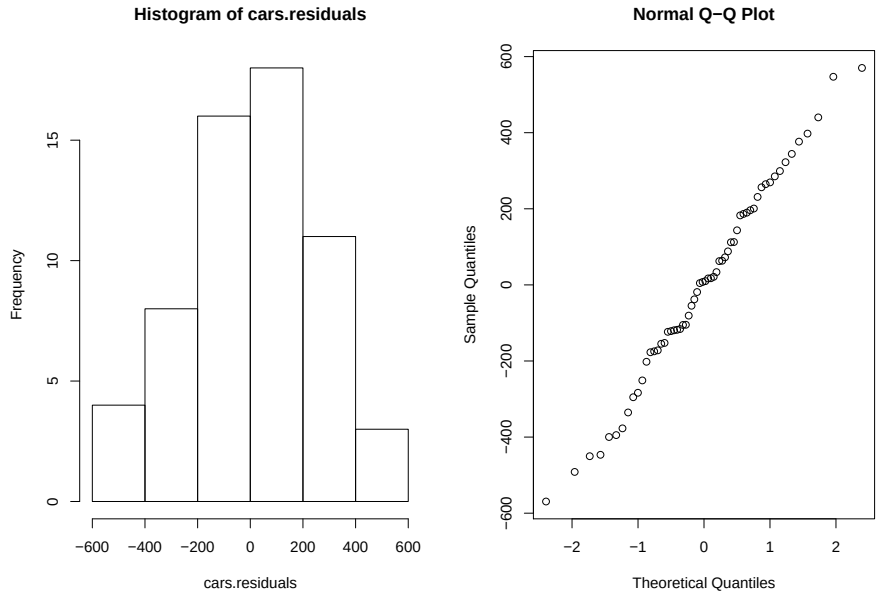

Figure 8.1: A histogram and a qq-plot of the model residuals to check normality of the residuals.

```
add1(cars.lm, Weight~Mileage+Disp.)
Single term additions

Model:
Weight ~ Mileage
       Df Sum of Sq     RSS     AIC
<none>              4078578     672
Disp.   1   1297541 2781037     651
```

`drop1` This function is used to see what the result is, in terms of sums of squares and residual sums of squares, of dropping a term (variable) from the model.

```
drop1(cars.lm, ~Mileage)
Single term deletions

Model:
Weight ~ Mileage
        Df Sum of Sq      RSS      AIC
<none>                4078578      672
Mileage  1  10428530 14507108      746
```

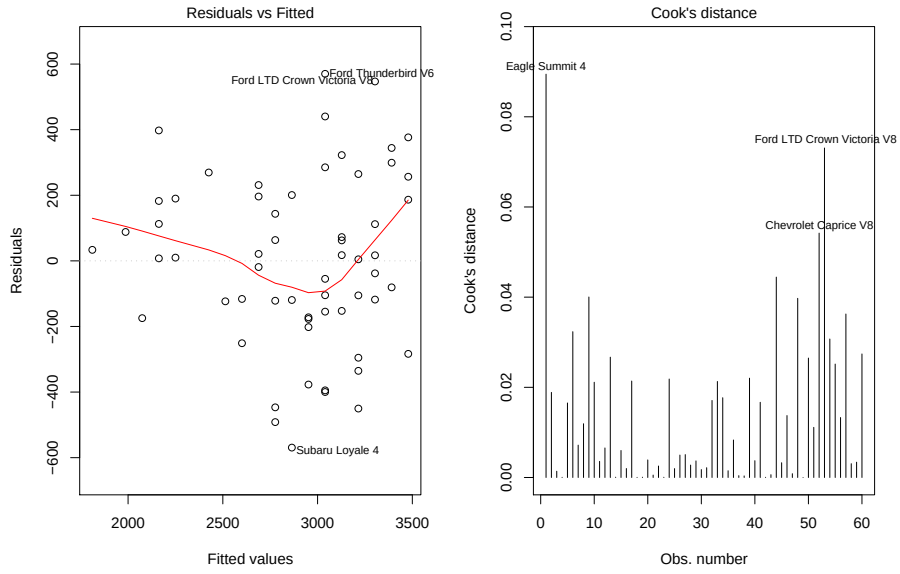

Figure 8.2: Diagnostic plots to check for linearity and for outliers.

`update` This function is used to update a model. In contrary to `add1` and `drop1` this function returns an object of class 'lm'. The following call updates the cars.lm object. The ∼`.+Disp` construction adds the Disp. variable to whatever model is used in generating the cars.lm object.

```
cars.lm2 <-  update(cars.lm, ~. + Disp.)
cars.lm2

Call:
lm(formula = Weight ~ Mileage + Disp., data = cars)

Coefficients:
(Intercept)      Mileage        Disp.
   3748.444      -57.976        3.799
```

8.3.3 Multicollinearity

The linear regression model can be formulated in matrix notation as follows:

$$y = X\beta + \epsilon$$

where X has N rows, the number of observations and $p+1$ columns the number of regression coefficients plus an intercept. Then for a normally distributed error term ϵ it can be shown that the least squares estimates $\hat{\beta}$ for the parameter β are given by

$$\hat{\beta} = (X'X)^{-1}X'y \tag{8.1}$$

When the matrix X does not have full rank, so less then $p+1$, then the matrix $X'X$ in equation 8.1 is singular and an inverse does not exists. This is the case of perfect multicollinearity, which does not happen often in practice. The problem of nearly perfect multicollinearity occurs when $X'X$ is nearly singular. This occurs when two or more regression variables are strongly correlated. Consider the following simulated data.

```
x1 <- runif(100,1,2)
x2 <- runif(100,1,2)
x3 <- 2*x1 +4*x2   + rnorm(100, 0, 0.01)
y <- 6*x1 + 5*x2 + 3*x2 + rnorm(100, 0, 0.4)

testdata <- data.frame(y,x1,x2,x3)
out.model <- lm(y ~ x1+x2+x3, data = testdata)
summary(out.model)
Call:
lm(formula = y ~ x1 + x2 + x3, data = testdata)

Residuals:
     Min       1Q   Median       3Q      Max
-1.09211 -0.26002  0.05173  0.29653  0.82532

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.1932     0.3146  -0.614    0.541
x1            0.7615     8.3111   0.092    0.927
x2           -2.4102    16.5890  -0.145    0.885
x3            2.6310     4.1428   0.635    0.527

Residual standard error: 0.4079 on 96 degrees of freedom
Multiple R-Squared: 0.9815,Adjusted R-squared: 0.9809
F-statistic:  1698 on 3 and 96 DF,  p-value: < 2.2e-16
```

Looking at the output, a strange thing is the huge standard error for `x2`. This may indicate that there is something wrong.

SVD and VIF

Two tools to detect multicollinearity are the *singular value decomposition* (SVD) of the X matrix and the calculation *variance inflation factors* (VIF).

The singular value decomposition of X finds matrices U, D and V such that

$$X = U \begin{pmatrix} d_0 & & & \\ & d_1 & & \\ & & \ddots & \\ & & & d_p \end{pmatrix} V$$

When X does not have full rank, one or more of the singular values d_i are zero. In practice this will not happen often. The more likely case is that the smallest singular value is small compared to the largest singular value. The SVD of the X matrix in the above example can be calculated with the function `svd`.

```
X <- model.matrix(out.model)
svd(X)
$d
[1] 94.26983374  3.06760623  1.29749565  0.02145514
... matrix U and V not displayed...
```

The variance inflation factors $VIF_i, i = 1, ..., p$ are based on regressing one of the regression variables x_i on the remaining regression variables $x_j, j \neq i$ for $i = 1, ..., p$. For each of these regressions the R-squared statistic $R_i^2, i = 1, ..., p$ can be calculated. Then the VIF is defined as

$$VIF_i = \frac{1}{1 - R_i^2}$$

It can be shown that the VIF_i can be interpreted as how much the variance of the estimated regression coeeficient β_i is inflated by the existence of correlation among the regression variables in the model. A VIF_i of 1 means that there is no correlation among the i-th regression variable and the remaining regression variables, and hence the variance of β_i is not inflated at all. The general rule of thumb is that VIFs exceeding 4 warrant further investigation, while VIFs exceeding 10 are signs of serious multicollinearity requiring correction.

The function `vif` in the 'DAAG' package calculates the VIFs for a fitted linear regression model.

```
library(DAAG)
vif(out.model)
    x1      x2      x3
 4150.9 13130.0 17797.0
```

8.3.4 Factor (categorical) variables as regression variables

The `lm` function (and other modeling functions, such as `coxph` and `glm`, as well) accepts factor variables (categorical variables) as regression variables. It is not possible to estimate a parameter for each level of the factor variable, as the model could be overparameterized, and the $X'X$ matrix in formula 8.1 would be singular. One can avoid overparameterization by using a so-called contrast matrix to impose restrictions on the parameters. By default R uses the so called *treatment contrast matrix* for unordered factor variables, and a *polynomial contrast matrix* is used for ordered factor variables. There are other contrasts.

The estimated parameter values in a treatment contrast are easy to interpret. One factor level is left out, then the parameter values of the other levels represent the difference between that level and the level that is left out.

Consider the following example where we create an artificial data frame with one numeric response column and one factor column with four levels (A,B,C and D) as the regression variable. The corresponding `y` values of each factor level have a certain mean as calculated in the code below.

```
y1 <- rnorm(100) +  5
y2 <- rnorm(100) + 10
y3 <- rnorm(100) + 30
y4 <- rnorm(100) + 50
y <- c(y1, y2, y3, y4)
x <- as.factor(
    c(rep("A",100),
      rep("B",100),
      rep("C",100),
      rep("D",100))
)
testdata <- data.frame(x, y)
lm(y~x, data=testdata)

Call:
lm(formula = y ~ x, data = testdata)

Coefficients:
(Intercept)           xB           xC           xD
      4.930        5.177       25.107       45.087
```

Level A has been left out. You can see that parameter value `xC` is about 25, representing the difference in mean between level A and level C.

When using a treatment contrast, the lowest level is left out of the regression. By default, this is the level with the name that comes first in alphabetical order. The parameters estimates for the remaining levels represent the difference between that level and the lowest level.

Consider the above example code again, but rename level A to level X. and fit the linear model again.

```
x <- as.factor(c(rep("X",100),rep("B",100),rep("C",100),rep("D",100)))
testdata <- data.frame(x,y)
lm(y~x,data=testdata)

Call:
lm(formula = y ~ x, data = testdata)

Coefficients:
(Intercept)           xC           xD           xX
     10.107       19.930       39.910       -5.177
```

Now level B is lowest level and is left out. So the parameter estimate for xC represents the difference in mean between level B and C, which is about 20.

If you are using a treatment contrast, the lowest level will be left out. When you don't want to leave out that particular level, you can use the so-called SAS contrast. This is the treatment contrast but leaving out the last factor level.

```
lm(y~x, data=testdata, contrasts = list(x = contr.SAS))
Call:
lm(formula = y ~ x, data = testdata, contrasts = list(x = contr.SAS))

Coefficients:
(Intercept)           x1           x2           x3
      4.930        5.177       25.107       45.087
```

Or alternatively you can reorder the factor. Suppose you want to leave out level C in the above example, proceed as follows.

```
testdata$x <- ordered(testdata$x,levels=c("C","B","D","X"))
```

The order in the levels is specified by the levels argument. You can check the order by printing the levels of the variable.

```
levels(testdata$x)
[1] "C" "B" "D" "X"

lm(y~x, contrast=list(x=contr.treatment), data=testdata)

Call:
lm(formula = y ~ x, data = testdata, contrasts = list(x = contr.treatment))

Coefficients:
(Intercept)           x2           x3           x4
      30.04       -19.93        19.98       -25.11
```

In the above example, we used the function `ordered` to define the level C as the lowest level. Consequently, level C is left out in the regression and the remaining parameters are interpreted as difference between that level and the level C.

The `reorder` function is used to order factor variables based on some other data. Suppose we want to order the levels of `x` in such a way that the lowest level has the smallest variance in `y`, then we use `reorder` as follows:

```
testdata$x <- reorder(testdata$x,testdata$y,var)
levels(testdata$x)
[1] "D" "X" "C" "B"
```

Level D has the smallest variance in `y`, and will be left out in a regression where we use a treatment contrast for the regression variable `x`.

8.4 Logistic regression

Logistic regression can be used to model data where the response variable is binary, a factor variable with two levels. For example, a variable Y with two categories 'yes' / 'no' or 'good' / 'bad'. Many companies have build score cards based on logistic regression where they try to separate 'good' customers from 'bad' customers. The logistic regression model calculates the probability of an outcome, $P(Y = good)$ and $P(Y = bad) = 1 - P(Y = good)$ given some regression variables $X_1, ..., X_p$ as follows:

$$P(Y = good) = \frac{\exp(\alpha_0 + \alpha_1 X_1 + ... + \alpha_p X_p)}{1 + \exp(\alpha_0 + \alpha_1 X_1 + ... + \alpha_p X_p)} \tag{8.2}$$

The regression coefficients $\alpha_0, ..., \alpha_p$ are estimated with a data set.

8.4.1 The modeling function `glm`

The function `glm` can be used to fit a logistic regression model. Let's generate a data set and demonstrate the different aspects of building a logistic regression model.

```
nrecords = 1000
ncols = 3
x <- matrix( runif( nrecords*ncols), ncol=ncols)
y <-  2 + x[,1] + 3*x[,2] - 4*x[,3]
y = exp(y)/(1+exp(y))
ppp <- runif(nrecords)
y <- ifelse(y > ppp, "good", "bad")
testdata <- data.frame(y, x)
```

To get an idea which variables in your data set have an influence on your binary response variable: plot the observed fraction ('yes'/'no') against the (potential) regression variables.

- Divide the variable X_i into say ten buckets (equal intervals).
- For each bucket calculate the observed fraction 'good'/ 'bad'.
- Plot bucket number against observed fraction.

Some R code that plots observed fractions for a specific regression variable:

```
obs.prob <- function(x)
{
  ## observed fraction of good, the
  ## second level in this case
  out <- table(x)
  out[2]/length(x)
}

plotfr <- function(y, x, n=10)
{
  tmp <- cut(x,n)
  p <- tapply(y,tmp, obs.prob)
  plot(p)
  lines(p)
  title(
     paste(deparse(substitute(y)),
     "and",
     deparse(substitute(x)))
  )
}
```

```
par(mfrow=c(2,2))
plotfr(testdata$y, testdata$X1)
plotfr(testdata$y, testdata$X2)
plotfr(testdata$y, testdata$X3)
```

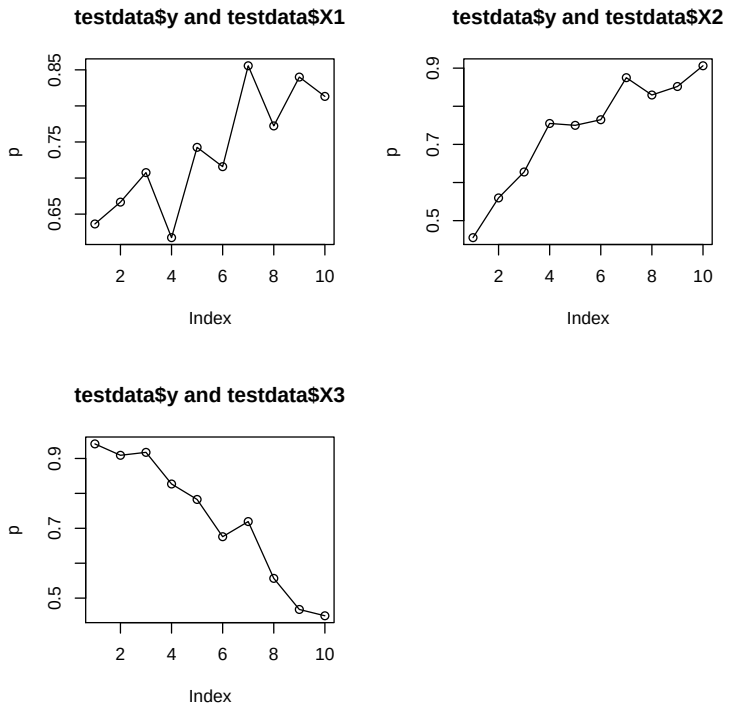

Figure 8.3: Explorative plots giving a first impression of the relation between the binary `y` variable and `x` variables.

The plots in figure 8.3 show strong relations. For variable `X3` there is a negative relation, just as we have simulated. The interpretation of the formula object that is needed as input for `glm`, is the same as in `lm`. So for example the : operator is also used here for specifying interaction between variables. The following code fits a logistic regression model and stores the output in the object `test.glm`.

```
test.glm = glm(y ~ X1 + X2 + X3, family = binomial, data=testdata)
summary(test.glm)
Call:
glm(formula = y ~ X1 + X2 + X3, family = binomial, data = testdata)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.5457  -0.5829   0.3867   0.6721   1.9312
```

```
Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   0.9616     0.2617   3.675 0.000238 ***
X1            1.6361     0.3065   5.338  9.4e-08 ***
X2            3.3955     0.3317  10.236  < 2e-16 ***
X3           -4.0446     0.3513 -11.515  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 1158.48  on 999  degrees of freedom
Residual deviance:  863.47  on 996  degrees of freedom
AIC: 871.47

Number of Fisher Scoring iterations: 5
```

The object `test.glm` is a 'glm' object. As with lm objects in the previous section, the glm object contains more information. Enter `print.default(test.glm)` to see the entire object. The functions listed in table 8.3 can also be used on glm objects.

8.4.2 Performance measures

To assess the quality of a logistic regression model several performance measures can be calculated. In the ROCR package there are functions to calculate the *Receiver Operator curve* (ROC), the Area under the ROC and lift charts from any glm model that is fitted.

A logistic regression model can be used to predict 'good' or 'bad'. Since the model only calculates probabilities of 'good', a threshold $t \in (0, 1)$ is chosen, if the probability is above t then 'good' is predicted otherwise bad is predicted. Dependent on this threshold t we will have the following numbers, TP, FP, FN and TN as displayed in the following so-called confusion matrix.

		Observed	
		Good	Bad
Model predicted	Good	TP	FP
	Bad	FN	TN
		number of goods	number of bads

Table 8.4: confusion matrix

Let n_g be the number of observed goods and n_b the number of observed bads, then we have:

1. TP (true positive) is the number of observations for which the model predicted good and that were observed good. True positive rate, $TPR = TP/n_g$.
2. TN (true negative) is the number of observations for which the model predicted bad and that were observed bad. True negative rate, $TNR = TN/n_b$.
3. FP (false positive) is the number of observations for which the model predicted good but were observed bad. False positive rate, $FPR = 1 - TNR$.
4. FN (false negative) is the number of observations for which the model predicted bad but were actually observed good. False negative rate, $FNR = 1 - TPR$.

The ROC curve is a parametric curve, for all tresholds $t \in (0, 1)$ the points (TPR, FPR) are calculated. Then these points are plotted. A ROC curve demonstrates several things:

1. It shows the trade-off between sensitivity and specificity (any increase in sensitivity will be accompanied by a decrease in specificity).
2. The closer the curve follows the left-hand border and then the top border of the ROC space, the more accurate the test.
3. The closer the curve comes to the 45-degree diagonal of the ROC space, the less accurate the test. The area under the curve (AUR) is a measure of how accurate the model can predict 'good'. A value of 1 is a perfect predictor while a value of 0.5 is very bad predictor.

The code below shows how to create an ROC and how to calculate the AUROC.

```
library(ROCR)
pred <- prediction( test.glm$fitted, testdata$y)
perf <- performance(pred, "tpr", "fpr")
plot(perf,  colorize=T, lwd= 3)
abline(a=0,b=1)
performance(pred, measure="auc")@y.values
[[1]]
[1] 0.8353878
```

8.4.3 Predictive ability of a logistic regression

Another way to look at the quality of a logistic regression model is to look at the predictive ability of the model. When we predict $P(Y = good)$ with the model, a pair of observations with different observed responses (one 'good', the other 'bad') is said to be *concordant* if the observation with the 'bad' has a lower predicted probability than the observation with the 'good'. If the observation with the 'bad' has a higher predicted

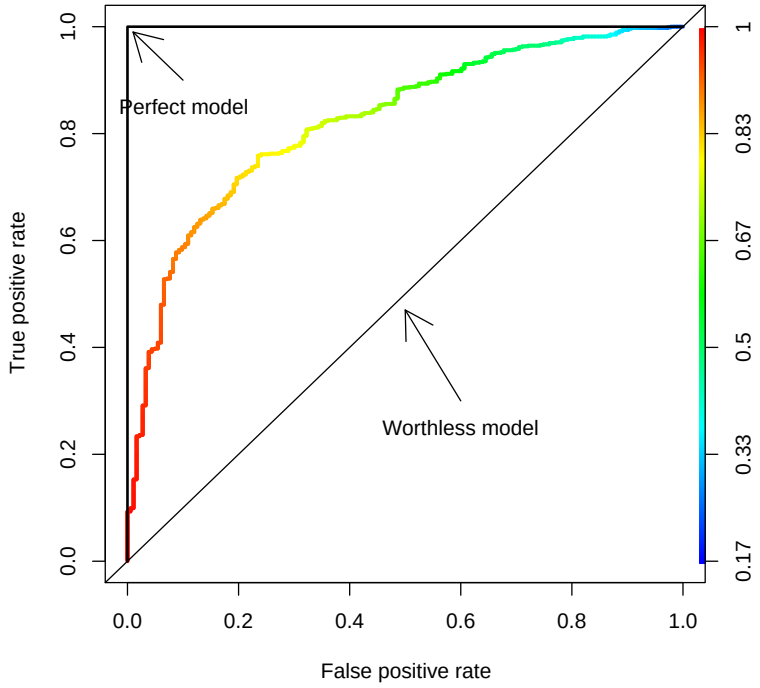

Figure 8.4: The ROC curve to assess the quality of a logistic regression model

probability than the observation with the 'good', then the pair is *discordant*. If the pair is neither concordant nor discordant, it is a tie.

Four measures of association for assessing the predictive ability of a model are available. These measures are based on the number of concordant pairs, n_c, the number of discordant pairs, n_d, let the total number of pairs t, and the number of observations, N.

1. The measure called c, also an estimate of the area under ROC,

$$c = (n_c + 0.5 \times (t - n_c - n_d))/t$$

2. Somer's D, $D = (n_c - n_d)/t$
3. Kendall's tau-α, defined as $(n_c - n_d)/(0.5 \cdot N \cdot (N-1))$
4. Goodman-Kruskal Gamma, defined as $(n_c - n_d)/(n_c + n_d)$

Ideally, we would like n_c to be very high and n_d very low. So the larger these measures the better the predictive ability of the model. The function `lrm` in the 'Design' package can calculate the above measures.

```
library(Design)
lrm(y ~ X1 + X2 + X3, data=testdata)
Logistic Regression Model

lrm(formula = y ~ X1 + X2 + X3, data = testdata)


Frequencies of Responses
bad good
198 802

 Obs  Max Deriv Model L.R.  d.f.     P       C     Dxy   Gamma   Tau-a      R2   Brier
1000      2e-08      233.9     3     0   0.824   0.649    0.65   0.206   0.331    0.12

          Coef    S.E.   Wald Z P
Intercept  1.9018 0.3089   6.16 0.0000
X1         0.8474 0.3147   2.69 0.0071
X2         3.1342 0.3403   9.21 0.0000
X3        -3.9718 0.3784 -10.50 0.0000
```

8.5 Tree models

Tree-based models are not only used for predictive modeling. They can be used to screening variables, assessing the adequacy of linear models and summarizing large multivariate data sets. When the response variable is a factor variable with two or more levels, then the term *classification tree* is used. The tree model produces rules like:

- IF Price $\leq$ 200 AND Weight $\leq$ 300 THEN Type is 'Small'.
- IF Price $>$ 200 AND Weight $>$ 300 AND Mileage $>$ 23 THEN Type is 'Van'.

When the response variable is numeric the tree is called a *regression tree*. The model produces rules like

- IF Price $\leq$ 200 AND Weight $\leq$ 300 THEN Mileage = 34.6.
- IF Price $>$ 200 AND Weight $>$ 456 AND Type is 'Van' THEN Mileage is 23.8.

These rules are constructed from the data by recursively dividing the data into disjoint groups by splitting certain variables. A detailed description of an algorithm is described in [10] and [11]. The basic ingredients of such an algorithm are:

- A measure for the quality of a split.
- A split selection rule. How and which variables do we split?
- A stopping criteria, we need to stop splitting at some stage before we end up with individual data points.

Compared to linear and logistic regression models trees have the following advantages

- Easier to interpret, especially when there is a mix of numeric and factor variables.
- Can model response variables that are factor and have more than two levels.
- More adept at capturing nonadditive behavior.

8.5.1 An example of a tree model

A nice feature of these rules is that they are easily visualized in a tree graph. There are some R packages that deal with trees. For example the packages rpart, party and randomForest. The latter two have methods to build multiple trees, that improve predictive accuracy compared to a single tree.

We consider the modeling function `rpart` in the 'rpart' package, that constructs a single tree. This package also contains the example data frame 'car.test.frame', we use it to construct a tree that predicts the type of a car given the mileage and price of that car.

```
library(rpart)
fit <- rpart(Type ~ Mileage + Price, data = car.test.frame)
## basic overview of the rules
fit
n= 60

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 60 45 Compact (0.25 0.05 0.22 0.22 0.15 0.12)
  2) Price>=9152.5 49 34 Compact (0.31 0.061 0.27 0.041 0.18 0.14)
    4) Mileage>=20.5 37 22 Compact (0.41 0.027 0.32 0.054 0.19 0)
      8) Mileage< 23.5 19  7 Medium (0.32 0.053 0.63 0 0 0) *
      9) Mileage>=23.5 18  9 Compact (0.5 0 0 0.11 0.39 0) *
    5) Mileage< 20.5 12  5 Van (0 0.17 0.083 0 0.17 0.58) *
  3) Price< 9152.5 11  0 Small (0 0 0 1 0 0) *

## detailed listing of an rpart object displaying only a part.
summary(fit)
Call:
rpart(formula = Type ~ Mileage + Price, data = car.test.frame)
  n= 60

         CP nsplit rel error    xerror       xstd
1 0.2444444      0 1.0000000 1.1555556 0.05851383
2 0.1555556      1 0.7555556 0.9555556 0.07756585
3 0.1333333      2 0.6000000 0.8444444 0.08294973
```

```
4 0.0100000      3 0.4666667 0.6000000 0.08563488

Node number 1: 60 observations,    complexity param=0.2444444
  predicted class=Compact  expected loss=0.75
    class counts:    15     3    13    13     9     7
   probabilities: 0.250 0.050 0.217 0.217 0.150 0.117
  left son=2 (49 obs) right son=3 (11 obs)
  Primary splits:
      Price   < 9152.5 to the right, improve=10.259180, (0 missing)
      Mileage < 27.5   to the left,  improve= 7.259083, (0 missing)
  Surrogate splits:
      Mileage < 27.5   to the left,  agree=0.933, adj=0.636, (0 split)
...
...
```

A graphical representation can be obtained with the following code

```
plot(fit)
text(fit)
```

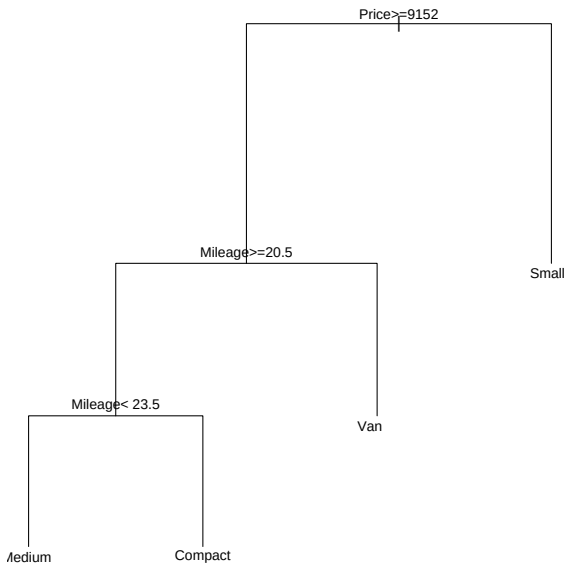

Figure 8.5: Plot of the tree: Type is predicted based on Mileage and Price

8.5.2 Coarse classification and binning

When building regression models *binning* or *coarse classification* is sometimes used.

Binning is a procedure that creates a nominal (factor) variable from a continuous (numeric) variable. I.e. each value of a numeric variable gets mapped to a certain interval (or category). There are a couple of reasons why we want to do this. First nonlinear effects can be captured in a very simple way, second the binned variable is less sensitive to outliers.

Coarse classification is a procedure to group all the possible outcomes of a nominal (factor) variable into a smaller set of outcomes. The main reason to do this is because there may be too many outcomes, and so some outcomes are very infrequently observed.

Tree based models can used in the regression context to create bins or perform the coarse classification. In this context a response variable and a regression variable for which we want to create bins are availabe. Suppose we have the following data:

```
age <- runif(500, 17, 75)
p = exp(-0.1*age) + 0.5
r <- runif(500)
y <- ifelse(p>r, "bad", "good")
testdata <- data.frame(age,y)
```

So the probability of observing 'good' increases with 'age'. For the creation of a score card we don't want to use the absolute value of age, we want to bin the age variable into bins and use those bins. How do we chose these bins?

- Simple approach, just 'manually' split the age variable into intervals. For example, intervals with the same number of points, or intervals with the same length.
- Use a tree based approach, so fit a tree with only the age variable as the regression variable and the 'good' /'bad' variable as the response.

To make the analysis more robust we want a minimum number of obervations in a bin, for example 30.

```
out <- rpart(
    y~age,
    data = testdata,
    control = rpart.control(minbucket= 30)
)
plot(out)
text(out)
```

The tree in Figure 8.6 shows the result of the binning. In this case the tree algorithm identifies four age intervals (bins): age < 26.6, $26.6 \leq$ age < 31.51, $31.51 \leq$ age < 41.88 and age ≥ 41.88 .

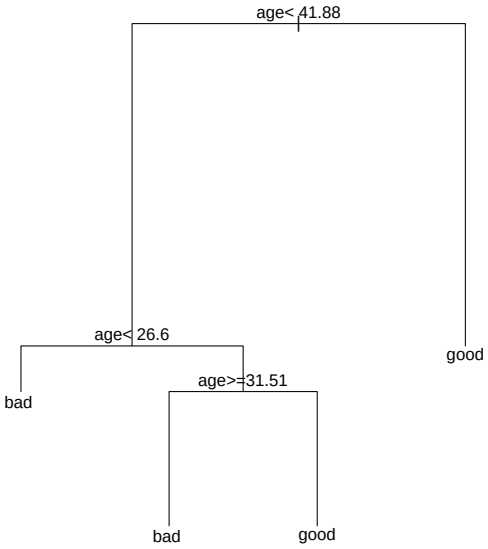

Figure 8.6: Binning the age variable, two intervals in this case

8.6 Survival analysis

In R survival analysis (also called churn analysis in marketing) of duration data (also called time to event data) can be done with either non-parametric approaches (Kaplan-Meier or Cox proportional hazards) or with parametric approaches (accelerated failure time models). Typical for duration data is (right)censoring. In a study the analyst can often not wait until all machines fail. At the time of study some machines may still work and the only information that we have is that the machine has survived a certain time span. This is called right censoring.

The R package 'survival' contains both non-parametric (the `coxph` function) and parametric modeling (the function `survreg`) functions. Another implementation of the accelerated failure time approach can be found in the package 'Design', the `psm` function. The modeling functions for survival analysis require the usage of an extra *packaging* function in the left hand side of a formula object. This function is used to indicate if a certain event was censored or not. For example, suppose we have a data frame with the columns time and status. The packaging function `Surv` connects time and status for right censored data. Where status = 1 means an event and status = 0 for censored.

```
Surv(time,status) ~ Age
```

The packaging function allows the user to specify a different type of censoring. For example, left censored data is specified as follows:

```
Surv(time,status, type="left") ~ Age + Sex
```

The right hand side of the formula has the same interpretation as in linear regression models.

8.6.1 The Cox proportional hazards model

To demonstrate the function `coxph` from the package 'survival', that fits a Cox proportional hazards model we use data that is analyzed in the paper of M. Prins and P.J Veugelers [12].

The data results from a multi-center cohort study among injecting drug users, one of the things the researches wanted to know was how long before a HIV infected person would develop AIDS, the incubation time. The data and a complete description can be downloaded from www.splusbook.com. In R we import the data into the `IDUdata` data frame and print only the first 10 persons of the data frame and show only a few columns. We also need to calculate the incubation time as the difference between the AIDS time and the entry time. The time unit used for the incubation time is months.

```
IDUdata <- read.csv("IDUdata.txt")
IDUdata$IncubationTime = IDUdata$AIDStime - IDUdata$Entrytime
> IDUdata[1:10, c(3,5,6,7,8,9,13)]
   Sex PosDate Age Entrytime AidsStatus Event IncubationTime
1    1     194  24       251          0     0             36
2    1     166  25       215          1     3             60
3    2     225  26       225          1     2             28
4    1     289  29       238          0     0             63
5    1     297  33       199          0     0            102
6    1     192  20       265          1     3             22
7    1     282  22       282          0     0             19
8    1     195  29       195          0     0            100
9    1     183  22       166          0     0            123
10   1     223  33       224          1     2             10
```

An estimation of the survival curve for the incubation time can be calculated with the function `survfit`.

```
kmfit <- survfit(Surv(IncubationTime, AidsStatus)~1, data=IDUdata)
kmfit
Call: survfit(formula = Surv(IncubationTime, AidsStatus) ~ 1, data = IDUdata)

      n  events  median 0.95LCL 0.95UCL
    418      76     135     118     Inf
```

The median survival time is estimated to be 135 months. So if a person is infected with HIV, he has a 50% probability that he will not develop AIDS within 135 months. A numerical and graphical output the complete survival curve can be created from the kmfit object. Use the functions `summary` and `plot`.

```
summary(kmfit)
Call: survfit(formula = Surv(IncubationTime, AidsStatus) ~ 1, data = IDUdata)

 time n.risk n.event survival std.err lower 95% CI upper 95% CI
    0    414       3    0.993 0.00417        0.985        1.000
    1    405       1    0.990 0.00483        0.981        1.000
    2    402       1    0.988 0.00541        0.977        0.998
    5    388       1    0.985 0.00596        0.974        0.997
    7    385       2    0.980 0.00694        0.967        0.994
   10    378       3    0.972 0.00821        0.956        0.989
 ...
 ...
plot(kmfit)
title("Survival curve for the AIDS incubation time (months)")
abline(h=c(0.9,0.8), lty=2)
abline(v=c(45,76), lty=2)
```

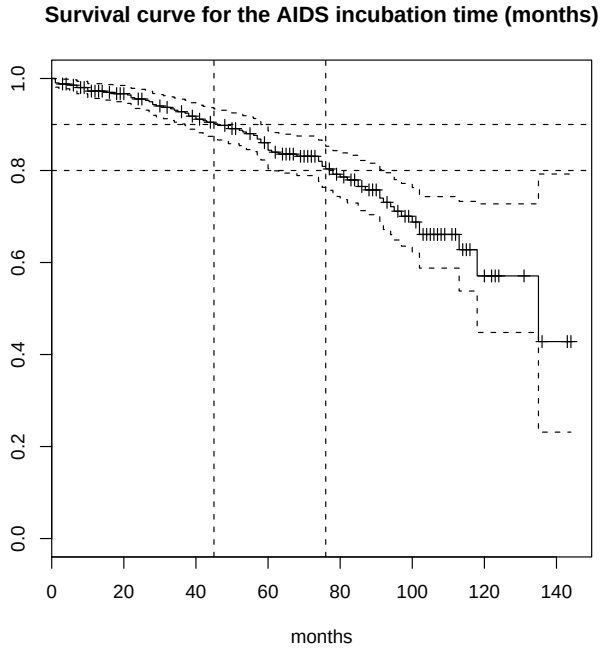

Figure 8.7: Survival curve: 10% will develop AIDS before 45 months and 20% before 76 months.

It is interesting to know if the age of a person has any impact on the incubation time. A Cox proportional hazards model is used to investigate that.

```
IDU.analysis1 <- coxph(
       Surv(IncubationTime, AidsStatus) ~ Age,
       data=IDUdata
)
```

The result of `coxph` is an object of class 'coxph'. It has its own printing method:

```
IDU.analysis1

Call:
coxph(formula = Surv(IncubationTime, AidsStatus) ~ Age, data = IDUdata)


      coef exp(coef) se(coef)    z    p
Age 0.0209      1.02   0.0175 1.20 0.23

Likelihood ratio test=1.39  on 1 df, p=0.238  n= 418
```

The `summary` function for 'coxph' objects returns the following information:

```
summary(IDU.analysis1)

Call:
coxph(formula = Surv(IncubationTime, AidsStatus) ~ Age, data = IDUdata)

  n= 418
      coef exp(coef) se(coef)    z    p
Age 0.0209      1.02   0.0175 1.20 0.23

    exp(coef) exp(-coef) lower .95 upper .95
Age      1.02       0.98     0.987      1.06

Rsquare= 0.003   (max possible= 0.851 )
Likelihood ratio test= 1.39  on 1 df,   p=0.238
Wald test            = 1.43  on 1 df,   p=0.231
Score (logrank) test = 1.44  on 1 df,   p=0.231
```

Use the generic function `resid` to extract model residuals. In a survival analysis there are several types of residuals, for example martingale residuals and deviance residuals. The residuals can be used assess the linearity of a regression variable or to identify

influential points. See [9] and [13] for a detailed discussion on how to use the residuals from a Cox model.

As an example we use the martingale residuals to look at the functional form of the Age regression variable. Do this by

- Fitting a model without the Age variable (in our case, the model reduces to a model with only the intercept).
- Extract the martingale residual from that model.
- Plot the martingale residual against the 'Age' variable, see Figure 8.8.

```
IDU.analysis0 <- coxph(Surv(IncubationTime,AidsStatus) ~ +1 , data=IDUdata)
mgaleres <- resid(IDU.analysis0, type="martingale")
plot(IDUdata$Age, mgaleres, xlab="Age", ylab="Residuals")
```

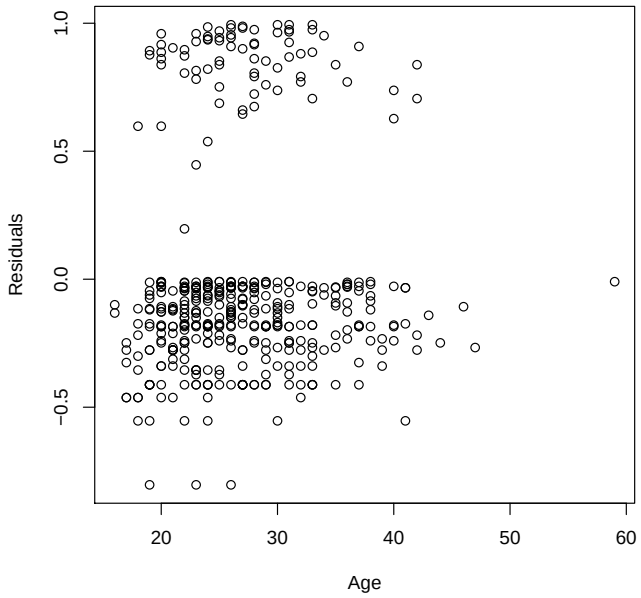

Figure 8.8: Scatter plot of the martingale residuals

An estimation of the survival time can be made for subjects of certain ages, use the function `survfit` as in the following code. The output shows that the median predicted survival time for a subject of age ten is infinite. As Figure 8.9 shows, the solid line corresponding to a subject of age ten never reaches 0.5.

```
newAges = data.frame(Age=c(10,30,60))
pred <- survfit(IDU.analysis1,newdata=newAges, se=T)
pred
      n events median 0.95LCL 0.95UCL
[1,] 418     76    Inf     135     Inf
[2,] 418     76    135     118     Inf
[3,] 418     76     97      60     Inf
plot(pred,lty=1:3)
```

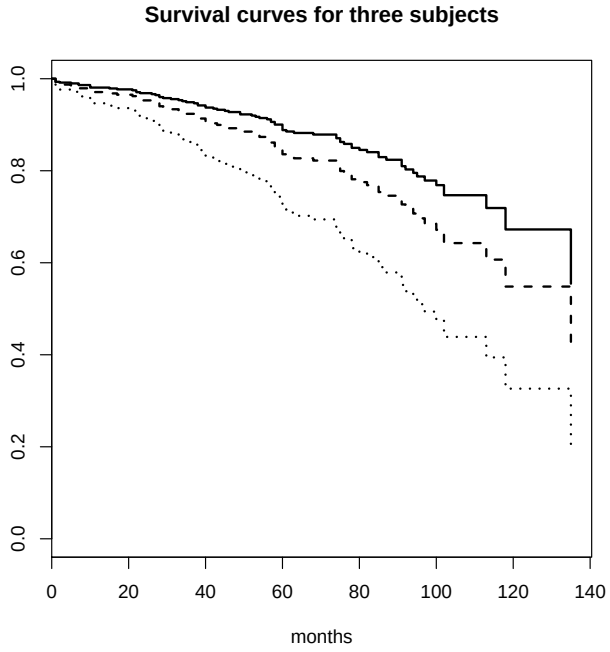

Figure 8.9: Three subjects with age 10, 30 and 60

8.6.2 Parametric models for survival analysis

In a parametric modeling approach the distribution of the time T to an event is modeled with a parametric model. For example a log normal or Weibull distrbution. The general form resembles the ordinary linear regression model and is given by:

$$f(T) = a_0 + a_1 X_1 + \cdots + a_p X_p + \sigma W$$

for some distribution W. A possible choice for f is the log function, this corresponds to the accelerated failure time model. The code below fits an accelerated failure time model for the IDU data with a Weibul distribution.

```
## some times are negative for convenience we set it to 1.
IDUdata$IncubationTime[IDUdata$IncubationTime <= 0] = 1
IDU.param <- survreg(
    Surv(IncubationTime, AidsStatus) ~ Age,
    data = IDUdata,
    dist = "weibull"
)
summary(IDU.param)

Call:
survreg(formula = Surv(IncubationTime, AidsStatus) ~ Age, data = IDUdata,
    dist = "weibull")
              Value Std. Error     z        p
(Intercept)  5.7839     0.3934 14.70 6.38e-49
Age         -0.0135     0.0132 -1.02 3.07e-01
Log(scale)  -0.2721     0.1001 -2.72 6.57e-03

Scale= 0.762

Weibull distribution
Loglik(model)= -510.6   Loglik(intercept only)= -511.1
Chisq= 1.02 on 1 degrees of freedom, p= 0.31
Number of Newton-Raphson Iterations: 7
n= 418
```

Predictions of the survival time can be made with the `predict` method.

```
newAges = data.frame(Age=30:35)
newAges$prediction = predict(IDU.param, newdata=newAges)
newAges
  Age prediction
1  30   216.6348
2  31   213.7250
3  32   210.8543
4  33   208.0222
5  34   205.2282
6  35   202.4716
```

8.7 Non linear regression

A good book on nonlinear regression is [14]. The function `nls` in R can fit nonlinear regression models of the form:

$$y = f(x, \theta) + \epsilon$$

for some nonlinear function f and where x is a vector of regression variables. The error term ϵ is often assumed to be normally distributed with mean zero. The p unknown parameters are in the vector $\theta = (\theta_1, ..., \theta_p)$ and need to be estimated from data points $(y_i, x_i), i = 1, ..., n$.

Unlike the formula specification in linear models, the operators in a formula object for nonlinear models have the 'normal' mathematical meaning. For example, to specify the following nonlinear model:

$$y = \frac{\beta_1 x_1}{\beta_2 + x_2} + \epsilon$$

use the formula object:

```
y ~ b1*x1/(b2+x2)
```

The right hand side of an the formula for nonlinear models can also be a function of the data and parameters. For example:

```
mymodel <- function(b1,b2,x1,x2){
   b1*x1/(b1+x2)
}

y ~ mymodel(b1,b2,x1,x2)
```

The `nls` function tries to estimate parameters for a nonlinear model that minimize the sum of squared residuals. So the following statement:

```
nls(y ~ mymodel(b1,b2,x1,x2))
```

minimizes

$$\sum_i (\texttt{y[i]} - \texttt{mymodel(b1,b2,x1[i],x2[i])})^2$$

with respect to `b1` and `b2`. In nonlinear models the right hand side of the model formula may be empty, in which case R will minimize the sum of the quadratic right hand side terms. The above specification, for example, is equivalent to

```
nls(~ y - mymodel(b1,b2,x1,x2))
```

To demonstrate the `nls` function we first generate some data from a known nonlinear model and add some noise to it.

```
x <- runif(100,0,30)
y <-  3*x/(8+x)
y <- y +rnorm(100,0,0.15)
our.exp <- data.frame (x=x,y=y)
```

The next plot is a scatterplot of our simulated data.

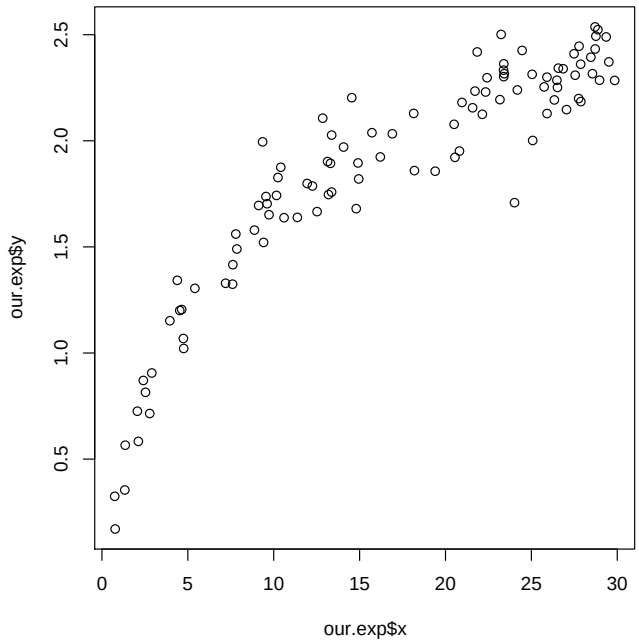

Figure 8.10: Scatter plot of our simulated data for `nls`

The model that we used to simulate our example data is the so-called Michaelis-Menten model, which is given by the following form:

$$y = \frac{\beta_1 x}{\beta_2 + x} + \epsilon$$

where ϵ is normally distributed, β_1 has value 3 and β_2 has value 8. To fit the model and display the fit results, proceed as follows:

```
fit1 <- nls(
    y ~ beta1*x /(beta2 + x),
    start = list( beta1 = 2.5, beta2 = 7)),
    data=our.exp
```

```
)
summary(fit1)
Formula: y ~ beta1 * x/(beta2 + x)

Parameters:
      Estimate Std. Error t value Pr(>|t|)
beta1   2.9088     0.0672   43.28   <2e-16 ***
beta2   7.3143     0.5376   13.61   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1414 on 98 degrees of freedom
```

So the first argument of `nls` is a formula object. Unlike the formula specification in lm or coxph the operators here have the 'normal' mathematical meaning. The second argument is required and specifies the initial values of the parameters. They are used to initiate the optimization algorithm to estimate the parameter values. The third argument is the data frame with the data.

Note that the nls function can sometimes fail to find parameter estimates. One of the reasons could be poor initial values for the parameters. For example:

```
fit1 <- nls(
    y ~ a*x/(b+x),
    start = list(a = 25000, b = 600),
    data = our.exp
)
Error in
    nls(
        y ~ a * x/(b + x),
        start = list(a = 25000, b = 600),
        data = our.exp
    ):
singular gradient
```

The output of `nls` is a list of class 'nls'. Use the generic `summary` function to get an overview of the fit. The output of summary is also a list, it can be stored and used to calculate the variance matrix of the estimated parameters. Although you can do this calculation directly on the components of the list, as in the code below.

```
fit1.sum <- summary(fit1)
fit1.sum$cov.unscaled * fit1.sum$sigma^2
            beta1      beta2
beta1 0.004516374 0.03408523
beta2 0.034085234 0.28903582
```

In this case it is more convenient to use the function `vcov` which is a generic function that also accepts a model object other than that generated by `nls`.

```
vcov(fit1)
            beta1      beta2
beta1 0.004516374 0.03408523
beta2 0.034085234 0.28903582
```

Use the function `predict` to calculate model predictions and standard errors of these predictions. Suppose we want to calculate prediction on the values of x from 0 to 10. Then we proceed as follows:

```
x  <- seq(0,30,l=100)
pred.data <- data.frame(x=x)
x.pred <- predict(fit1, newdata = pred.data)
```

The output object `x.pred` is a vector which contains the predictions. You can insert the predictions in the `pred.data` data frame and plot the predictions together with the simulated data as follows:

```
pred.data$ypred <- x.pred
plot(our.exp$x, our.exp$y)
lines(pred.data$x,pred.data$ypred)
```

8.7.1 Ill-conditioned models

Similair to the problem of multicollinearity in linear regression (as described in section 8.3.3), nonlinear models can be ill-conditioned too. However, with nonlinear models it may not only be a data issue, but the nonlinear model it self may be ill conditioned. Such a model can cause the estimation procedure to fail, or estimated model parameters may have very large confidence intervals. Consider the following model, the so-called Hill equation:

$$f(x,\theta) = V_m \frac{x^\alpha}{k^\alpha + x^\alpha}$$

Given the data points in Figure 8.12 we see that two sets of paramters fit the data equally well. The solid and dashed lines corresponds to $\alpha = 0.8, 3.1, V_m = 1.1.08, 1, k = 0.3, 1$ respectively. Either more data at lower x values are needed or a different model must be used.

The following R code simulates some data from the model and fits the model with the simulated data.

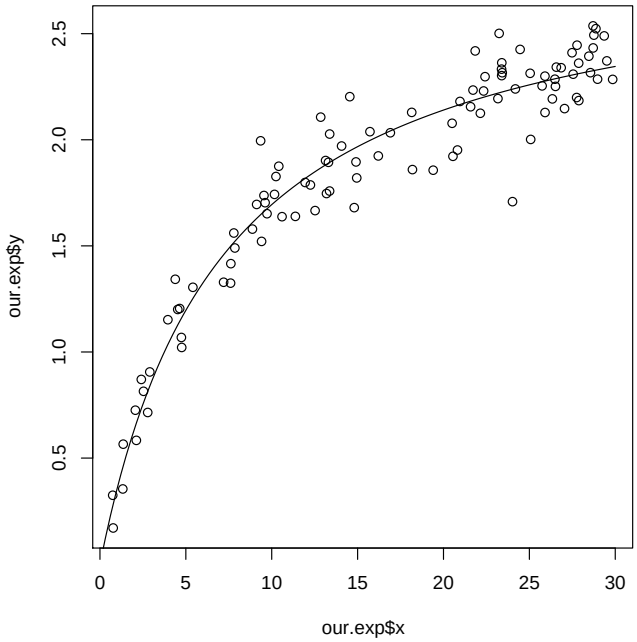

Figure 8.11: Simulated data and `nls` predictions

```
## Create the model function
HillModel <- function(x, alpha, Vm, k)
{
    Vm*(x^alpha)/(k^alpha + x^alpha)
}
## Simulate data and put it in a data frame
k1 = 0.3; Vm1 = 1.108; alpha1 = 0.8
x <- runif(45, 1.6, 5)
datap <- HillModel(x, alpha1,Vm1,k1) + rnorm(45, 0, 0.09)
simdata <- data.frame(x, datap)
## Fit the model
out <- nls(
    datap ~ HillModel(x, alpha,Vm,k),
    data = simdata,
    start = list( k = 0.3, Vm=1.108, alpha=0.8)
)
## Print output
summary(out)
vcov(out)
Formula: datap ~ HillModel(x, alpha, Vm, k)

Parameters:
      Estimate Std. Error t value Pr(>|t|)
k       0.1229     1.4140   0.087  0.93116
Vm      1.0108     0.3184   3.174  0.00281 **
alpha   0.9399     5.5311   0.170  0.86588
---
```

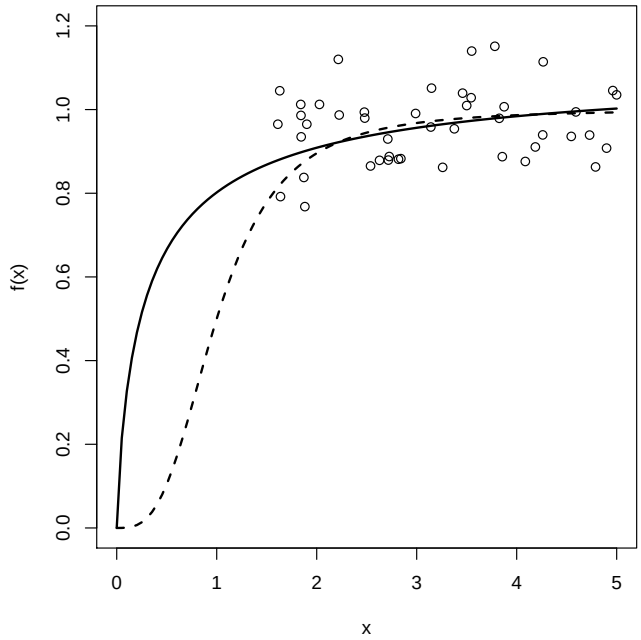

Figure 8.12: Hill curves for two sets of parameters

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

Residual standard error: 0.08745 on 42 degrees of freedom

Number of iterations to convergence: 14
Achieved convergence tolerance: 6.249e-06

               k         Vm    alpha
k      1.9993975 -0.4390971  7.79519
Vm    -0.4390971  0.1013896 -1.74292
alpha  7.7951900 -1.7429199 30.59291
```

Eventhough the fitting routine `nls` started with the same parameter values as those that were used in simulating the data the nls function does not get really close, and the standard error of the alpha parameter is quit large. Even more disturbing, when we simulate new data with the same parameters the `nls` function will come up with very different results. When observations with a smaller x value are available the problem is less ill-conditioned.

```
## simulate data with smaller x values
x <- runif(45, 0.01, 5)
datap <- HillModel(x, alpha1,Vm1,k1) + rnorm(45, 0, 0.09)
simdata <- data.frame(x, datap)
## Fit the model
out <- nls(
```

```
    datap ~ HillModel(x, alpha,Vm,k),
    data = simdata,
    start = list( k = 0.3, Vm=1.108, alpha=0.8)
)
## Print output
summary(out)
vcov(out)
      Estimate Std. Error t value Pr(>|t|)
k      0.24442    0.04727   5.171  6.1e-06 ***
Vm     1.04371    0.06781  15.392  < 2e-16 ***
alpha  0.94588    0.28034   3.374  0.00160 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.08786 on 42 degrees of freedom

Number of iterations to convergence: 4
Achieved convergence tolerance: 8.43e-07

                  k           Vm       alpha
k      0.002234067  0.001549951 -0.00304783
Vm     0.001549951  0.004598040 -0.01778873
alpha -0.003047830 -0.017788730  0.07859150
```

When data with a smaller x value are not available, the Hill model with three parameters is not identifyable. Maybe a parameter should be fixed at a certain value instead of trying to estimate it.

8.7.2 Singular value decomposition

A useful trick to find out if certain parameters or combination of parameters are estimable (identifyable), i.e. whether they influence model predictions enough, or whether they will be obscured by measurement noise, is a singular value decomposition of the so-called *sensitivity matrix*. Let us assume we have a (non linear) model.

$$y_i^{\text{pred}} = f(x_i, \theta).$$

For data point $i = 1, ..., n$. Then the sensitivity matrix $S(\theta)$ for the parameter vector θ, is defined by

$$S(\theta) = \frac{\partial y^{\text{pred}}}{\partial \theta}.$$

So S can be calculated by:

$$(S(\theta))_{ij} = \left(\frac{\partial y^{\text{pred}}}{\partial \theta} \right)_{ij} = \frac{\partial y_i^{\text{pred}}}{\partial \theta_j}.$$

This method will rank the importance with respect to the influence on y^{pred} of linear combinations of the parameters. Thereto a singular value decomposition of S is performed.

$$S = U \begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_p \end{pmatrix} V \ ,$$

where U and V are unitary, and the d_i are called the singular values of $S(\theta)$. This can also be interpreted as

$$\Delta(y^{\text{pred}}) \simeq U \begin{pmatrix} d_1 & & & \\ & d & & \\ & & \ddots & \\ & & & d_p \end{pmatrix} V \cdot \Delta\theta \ .$$

So the i-th singular value d_i shows the effect of changes of the parameters in the direction given in the i-th row of V. If a singular value drops below a certain critical value or is relatively small compared to the largest singualr value then the model shows signs of ill-conditioning. This certainly obvious if a singular value is (nearly) zero, a small change in the parameter will have no effect on the measurement space.

Note that for a linear regression model $y = X\beta$ the sensitivity matrix S is just the matrix X and that small singular values correspond to the multicollinearity problem, see section 8.3.3.

For the Hill model the code below uses the function `deriv` for the calculation of the sensitivity matrix and the function `svd` for its singular value decomposition.

```
## calc symbolic derivatives with respoect to the parameters
ModelDeriv  <- deriv(
    expression( Vm*(x^alpha)/(k^alpha + x^alpha)),
    name = c("Vm", "alpha", "k")
)
## evaluate the derivative at a certain x and parameter values
sensitivity <- eval(
    ModelDeriv,
    envir = list(
        x = seq(from = 1.6,to = 5, l = 50),
        k = 0.3,
```

```
        Vm = 1.108,
        alpha = 0.8
    )
)
## the gradient matrix is given as an attribute extract it and
## calculate the singular value decomposition
sensitivity <- attributes(sensitivity)$gradient
svd(sensitivity)
$u
...
matrix u skipped
...
$d
[1] 6.89792666 0.52385899 0.03482234
$v
           [,1]       [,2]       [,3]
[1,] -0.8862039 -0.4128734 -0.2101863
[2,] -0.3014789  0.1694330  0.9382979
[3,]  0.3517857 -0.8948900  0.2746248
```

The largest singular value is 6.898 and the smallest has a value of 0.0348. This ratio becomes better is we include data points with smaller x values.

```
sensitivity <- eval(
    ModelDeriv,
    envir = list(
        x = seq(from = 0.01, to = 5, l = 50),
        k = 0.3,
        Vm = 1.108,
        alpha = 0.8
    )
)
sensitivity <- attributes(sensitivity)$gradient
svd(sensitivity)
...
$d
[1] 6.6156912 1.3827939 0.4279231
...
```

9 Miscellaneous Stuff

9.1 Object Oriented Programming

9.1.1 Introduction

The programming language in R is object-oriented, In R this means:

- All objects in R are members of a certain class.
- There are *generic* methods that will pass an object to its *specific* method.
- The user can create a new classes, new generic and specific methods.

There are many classes in R, such as 'data.frame', 'lm' and 'h.test'. The function `data.class` can be used to request the class of a specific object.

```
mydf <- data.frame(x=c(1,2,3,4,5), y = c(4,3,2,1,1))
data.class(mydf)
[1] "data.frame"
myfit <- lm(y~x, data=mydf)
data.class(myfit)
[1] "lm"
```

There are two object oriented systems in R, old style classes (also called S version 3 or S3 classes), and new style classes (also called S version 4 or S4 classes). We first discuss old style classes and then new style classes. Note that many of the existing routines still make use of the old-style classes. When creating new classes, it is recommended to use new style classes.

9.1.2 Old style classes

Generic and specific methods

In R there are a number of generic methods that can be applied to objects. For example, any object can be printed by using the generic `print` method:

```
print(mydf)
print(myfit)
```

or simply

```
mydf
myfit
```

A data frame will be printed in a different way than an object of class lm. It may not be surprising that the generic `print` function does not do the actual printing, but rather looks at the class of an object and then calls the specific print method of this class. The function `print` therefore does not show much of the code that does the actual printing.

```
print
function (x, ...)
UseMethod("print")
<environment: namespace:base>
```

A generic function has this form, a 'one-liner' with a call to the function `UseMethod`. For example, if the class of the object `myfit` is lm then `print(myfit)` will call the function `print.lm`. If the class of the object is 'someclass' then R will look for the function `print.someclass`. If that function does not exists then the function `print.default` will be called.

The function `methods` returns all specific methods for a certain class:

```
> methods(class="lm")
 [1] add1.lm*            alias.lm*           anova.lm            case.names.lm*
 [5] confint.lm*         cooks.distance.lm*  deviance.lm*        dfbeta.lm*
 [9] dfbetas.lm*         drop1.lm*           dummy.coef.lm*      effects.lm*
[13] extractAIC.lm*      family.lm*          formula.lm*         hatvalues.lm
[17] influence.lm*       kappa.lm            labels.lm*          logLik.lm*
[21] model.frame.lm      model.matrix.lm     plot.lm             predict.lm
[25] print.lm            proj.lm*            residuals.lm        rstandard.lm
[29] rstudent.lm         simulate.lm*        summary.lm          variable.names.lm*
[33] vcov.lm*

   Non-visible functions are asterisked
```

The output of the function methods is a vector with the specific methods. So for the class lm we see that `plot.lm` is a specific method, so a we could use `plot(lm.object)`. Another specific method is `extractAIC.lm`. The AIC quantity for a linear regression model can be calculated as follows: Fit a linear regression model with the function `lm`. This results in an object of class lm. Then apply the generic function `extractAIC`, which will call the specific `extractAIC.lm` function.

```
cars.lm <- lm(Price~Mileage,data=cars)
extractAIC(cars.lm)
[1]   2.0000 967.2867
```

The AIC quantity can also be calculated for other models, such as the Cox proportional hazards model. For the model fitted in section 8.6 with the function `coxph`, we extract the AIC:

```
IDU.analysis1 <- coxph(
       Surv(IncubationTime, AidsStatus) ~ Age,
       data=IDUdata
)
extractAIC(IDU.analysis1)
[1]   1.0000 796.3663
```

The function `methods` can also be used to see which classes have an implementation of a specific method.

```
methods(generic.function="extractAIC")
[1] extractAIC.aov*          extractAIC.coxph*        extractAIC.coxph.penal*
[4] extractAIC.glm*          extractAIC.lm*           extractAIC.negbin*
[7] extractAIC.survreg*

   Non-visible functions are asterisked
```

Creating new classes

R allows the user to define new classes and new specific and generic methods in addition to the existing ones. The function class can be used to assign a certain class to an object. For example:

```
mymatrix <- matrix(rnorm(50^2),ncol=50)
class(mymatrix) <- "bigMatrix"
```

The object `mymatrix` is now a matrix of class 'bigMatrix' (whatever that may mean). The class bigMatrix does not have a lot of meaning yet, since it does not have any specific methods. We will write a number of specific methods for objects of class bigMatrix in the following section. Using the function `class` directly is not recommended. One could for instance run the following statements without any complaints or warnings

```
m2 <- matrix(rnorm(16),ncol=4)
class(m2) <- "lm"
```

However, `m2` is not a real lm object. If it is printed, R will give something strange. When an lm object is printed, the specific function `print.lm` is called. This function expects a proper lm object with certain components. Our object `m2` does not have these components.

```
m2
Call:
NULL

Warning messages:
1: $ operator is deprecated for atomic vectors, returning NULL in: x$call
2: $ operator is deprecated for atomic vectors, returning NULL in: object$coefficients
No coefficients
```

So it is recommended to use a so-called constructor function. To create an object of certain class use only the constructor function for that class. The constructor function can then be designed in such a way that it only returns a 'proper' object of that class. If you want an lm object use the function `lm`, which can act as a constructor function for the class lm. For our bigMatrix class we create the following constructor function:

```
bigMatrix <- function(m)
{
  if(data.class(m) == "matrix")
  {
    class(m) = "bigMatrix"
    return(m)
  }
  else
  {
    warning("not a matrix")
    return(m)
  }
}

m1 <- bigMatrix("ppp")
m2 <- bigMatrix( matrix(rnorm(50^2),ncol=50))
```

Defining new generic and specific methods

Two specific methods can be created for our bigMatrix class, `print.bigMatrix` and `plot.bigMatrix`. Printing a big matrix results in many numbers on screen. The specific print method for bigMatrix only prints the dimension and the first few rows and columns.

```
print.bigMatrix <- function(x, nr=3,nc=5)
{
  cat("Big matrix \n")
  cat("dimension ")
  cat(dim(x))
  cat("\n")
  print(x[1:nr, 1:nc])
}
m2
Big matrix
dimension 50 50
           [,1]       [,2]      [,3]       [,4]       [,5]
[1,] -0.7012566 -0.7327267 -0.706452 -0.2355600 -1.2577592
[2,]  1.6390825 -0.2999556 -1.131336 -0.2536510 -0.3878151
[3,]  0.8964895  0.2022080  1.379076 -1.7892237  0.9087716
```

The plot method displays the matrix as an image plot.

```
plot.bigMatrix <- function(x)
{
  image(1:ncol(x),1:nrow(x),x)
  title(
    paste("plot of matrix ",
    deparse(substitute(x)), sep="")
  )
}
plot(m2)
```

9.1.3 New Style classes

There is no formal description of an object of S3 class. It could be a matrix where the user (accidently) assigned the lm class to it. The new style classes in R allow the user to define a new class more formally than old style classes. The new style classes differ from the old style classes:

- All objects of a new-style class must have the same structure. This is not true for many old-style classes. The new-style classes have a more formal and tighter specification.
- The new class mechanism has greater uniformity than the old and the new engine has many more tools to make programming easier.
- Methods can be designed for data types (for example, vectors) that where not classes in the old engine.
- Class inheritance is more rigorous than in the old style classes.

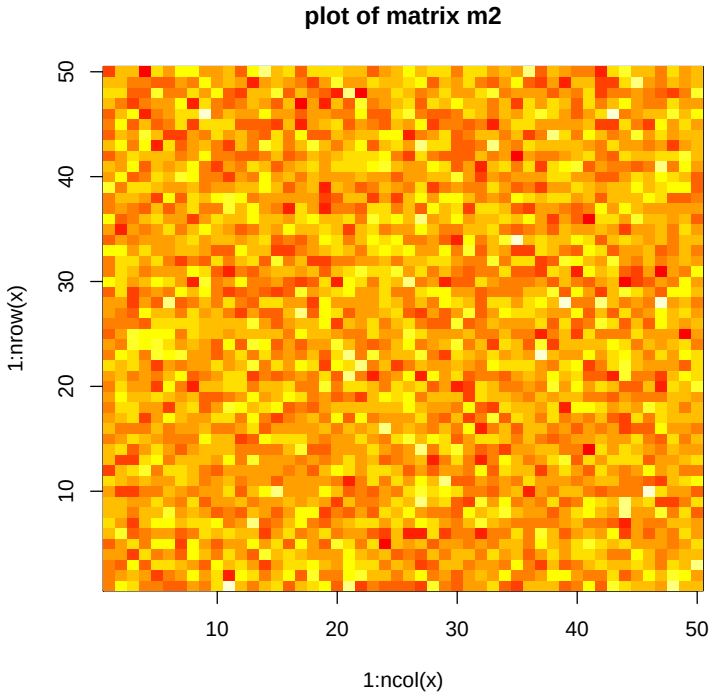

Figure 9.1: Result of the specific plot method for class bigMatrix.

Creating a new style class definition

New-style classes are made up of *slots*. These are similar to but distinct from components of a list, in that the number of slots and their names and classes are specified when a class is created: objects are extracted from slots by the `@` operator. Exact matching of slot names is used, unlike the `$` operator for lists.

A new style class definition is created with the function `setClass`. Its first argument is the name of the class, and its `representation` argument specifies the slot. For example, a class 'fungi' to represent the spatial location of fungi in a field might look like:

```
setClass("fungi",
   representation(
     x="numeric",
     y="numeric",
     species="character"
   )
)
```

Once a class definition has been created it can be examined by the function `getClass`.

```
getClass("fungi")
Slots:

Name:          x         y   species
Class:   numeric   numeric character
```

To list all the classes in R or in your workspace use the function `getClassses`.

```
## class definitions in your workspace
getClasses(where=1)
[1] "fungi"
```

The class 'fungi' can also be created by combining other classes. For example:

```
setClass("xyloc",
  representation(x="numeric", y="numeric")
)
setClass("fungi",
  representation("xyloc", species="character")
)
```

A class can be removed with the function `removeClass`. To create (or instantiate) an object from the fungi class use the function `new`.

```
field1 <- new("fungi",
    x=runif(10), y=runif(10),
    species=sample(letters[1:5],rep=T,10)
)
field1
An object of class "fungi"
Slot "x":
 [1] 0.41644379 0.89240433 0.88980142 0.77224325 0.80395122 0.83608564
 [7] 0.04149246 0.24511134 0.74946802 0.26268302

Slot "y":
 [1] 0.6828478 0.2134961 0.8681543 0.9748187 0.0253564 0.9479711 0.3381227
 [8] 0.3446705 0.4415452 0.0979566

Slot "species":
 [1] "a" "d" "e" "d" "b" "d" "c" "e" "a" "b"
```

When you instantiate new objects from a certain class, you can perform a validity check. For example, our 'fungi' class should have input vectors of the same lengths. We can build in a validity check as follows.

```
## function to check validity, it should return true or false
validFungi <- function(object){
  len <- length(object@x)
  if(len != length(object@y) || length(object@species) != len)
  {
    cat("Mismatch in length of slots")
    return(FALSE}
  }
else return(TRUE)
}

setClass("fungi",
    representation(
      x="numeric",
      y="numeric",
species="character"
    ),
    validity = validFungi
)
setValidity("fungi", validFungi)


field2 <- new("fungi",
    x=runif(110), y=runif(10),
    species=sample(letters[1:5],rep=T,110)
)
Error in validObject(.Object) : invalid class "fungi" object: FALSE
Mismatch in length of slots
```

The function `validFungi`, as any validity checking function, must have exactly one argument called `object`.

Creating new generic and specific methods

A generic function is `show`, which shows an object. If we want to show our objects from class fungi in a different way then we can write a new function (`myfungishow`) that shows our fungi object differently. The function `setMethods` sets the new function `myfungishow` as the specific show method for the fungi class. We have the following R code:

```
myfungishow <- function(object){
  tmp = rbind(
    x = format(round(object@x,2)),
    y = format(round(object@y,2)),
```

```
species = object@species
  )
  dimnames(tmp)[[2]] = rep("", length(object@x))
  print(tmp, quote=F)
}

setMethod("show","fungi", myfungishow)

field1

      x 0.97 0.55 0.44 0.03 0.92 0.46 0.49 0.92 0.30 0.19
      y 0.44 0.15 0.35 0.79 0.73 0.42 0.04 0.65 0.68 0.18
species c    e    a    c    c    d    e    e    e    b
```

Note that the `setMethod` function copies the function `myfungishow` into the class information. In fact after a call to `setMethod` the function `myfungishow` can be removed. This totally different from the old-style classes, where the specific method was searched for by a naming convention (print.fungi).

To see the specific show method for the fungi class use the function `getMethods`.

```
getMethods("show", w=1)
An object of class "MethodsList"
Slot "methods":
$fungi
Method Definition:

function (object)
{
    tmp = rbind(x = format(round(object@x, 3)), y = format(round(object@y,
        2)), species = object@species)
    dimnames(tmp)[[2]] = rep("", length(object@x))
    print(tmp, quote = F)
}

Signatures:
        object
target  "fungi"
defined "fungi"
...
...
```

9.2 R Language objects

9.2.1 Calls and Expressions

In R you can use the language to create new language constructions or adjust existing ones. For example, performing symbolic manipulation in R such as calculating derivatives symbolically, or writing functions that accept expressions as input. In R, 'language objects' are either objects of type 'name', 'expression' or 'call'. To deal with R language objects you should prevent the system from beginning the usual evaluation of expressions and calls. Suppose we have the following statements:

```
x <- seq(-3,3,l=100)
y <- sin(x)
```

The evaluated version of `sin(x)` is stored in the object `y` (so `y` contains 100 numbers). That this result originated from `sin(x)` is not visible from `y` anymore. To keep an unevaluated version, the normal evaluation needs to be interrupted. This can be done by the function `substitute`.

```
y <- substitute(sin(x))
y
sin(x)
```

The object `y` is a so called 'call' object. The object `y` can still be evaluated using the function `eval`.

```
eval(y)
  [1] -0.14112001 -0.20082374 -0.25979004 -0.31780241 -0.37464782
  [6] -0.48400786 -0.53612093 -0.58626538 -0.63425707 -0.67991980
 [11] -0.76359681 -0.80130384 -0.83606850 -0.86776314 -0.89627139
 ...
```

In order to print the object `y`, for instance in a graph, `y` must be deparsed first. This can be done using the function `deparse`, which converts `y` to a character object.

```
x <- seq(-3,3,l=100)
titletext <- deparse(substitute(sin(x))
y <- sin(x)
plot(x,y,type="l")
title(titletext)
```

This seems cumbersome, since we could simply have used:

```
title("sin(x)")
```

However, the substitute-deparse combination will come to full advantage in functions, for example:

```
printexpr <- function(expr){
  tmp <- deparse(substitute(expr))
  cat("The expression ")
  cat(tmp)
  cat(" was typed.")
  invisible()
}

printexpr(sin(x))
The expression sin(x) was typed.
```

The function `sys.call` can be used inside a function and stores the complete call to the function that contains the `sys.call` function.

```
plotit <- function(x,y){
  plot(x,y)
  title(deparse(sys.call())
}

plotit(rnorm(100),rnorm(100))
```

9.2.2 Expressions as Lists

Expressions and calls can be seen as recursive lists, so they can be manipulated in the same way you manipulate ordinary lists. To create calls use the function `quote`, to create expressions use the function `expression`. The output of these functions can be evaluated using the `eval` function.

```
my.expr <- expression(3*sin(rnorm(10)))
my.expr

eval(my.expr)
[1]  0.09410139  1.41480964  0.71378911  2.33318300  2.05434944  2.91265390
[7]  2.98911418  0.25474676 -2.02085040  0.88611195
```

Let's look at the expression `my.expr` we transform it to a list using the function `as.list`.

```
as.list(my.expr)
[[1]]:
3 * sin(rnorm(10))
```

This is a list with one component. Let us zoom in on this component and print it as a list.

```
as.list(my.expr[[1]])
[[1]]:
 '*'

[[2]]:
[1] 3

[[3]]:
sin(rnorm(10))
```

In this list, the first element of which is an object of class 'name'. Its second element is of class 'numeric' and its third element is of class 'call'. If we zoom in on the third element of the above list we get:

```
as.list(my.expr[[1]][[3]])
[[1]]:
sin

[[2]]:
rnorm(10)
```

Here the first component is of class 'name' and the second component is an object of class 'call'. Working with expressions in this way can be of use in case one has a function `testf` which calls another function that depends on calculations that occur inside `testf`, like in the following example:

```
testf <- function(){
  n <- rbinom(1,10,0.5)
  expr <- expression(rnorm(10))
  expr[[1]][[2]] <- n
  x <- eval(expr)
  x
}

testf()
```

Indeed this could have been achieved in a much simpler manner, such as in the code below. But it's the idea that counts here.

```
testf <- function(){
  n <- rbinom(1,10,0.5)
  x <- rnorm(n)
  x
}
```

9.2.3 Functions as lists

It may be surprising that we can also transform function objects to list objects. Lets look at a simple function.

```
myf <- function(x,y)
{
   temp1 = x+y
   temp2 = x*y
   tmp1/temp2
}
```

We use the function `as.list` to print the function as a list.

```
as.list(myf)
$x

$y

[[3]]
{
    temp1 = x + y
    temp2 = x * y
    tmp1/temp2
}
```

The result is a list with three components, when we transform the third component as a list we get:

```
as.list( as.list(myf)[[3]] )
[[1]]
`{`

[[2]]
temp1 = x + y

[[3]]
temp2 = x * y

[[4]]
tmp1/temp2
```

We can even go further, print the second component of the last list as a list.

```
as.list( as.list( as.list(myf)[[3]] )[[2]] )
[[1]]
`=`

[[2]]
temp1

[[3]]
x + y
```

9.3 Calling R from SAS

The SAS system provides many routines for data manipulations and data analysis. It may be hard to convince a 'long-time' SAS user to use R for data manipulation or statistics. However, the graphics in R are superior compared to what SAS/GRAPH can offer. Some graphs are unavailable in SAS/GRAPH or very time consuming to program.

We will give small examples on how to use R graphs in a SAS session.

9.3.1 The `call system` and `X` functions

In SAS there are two ways to call external programs, the `call system` and the `X` functions. The following example calls `Rcmd BATCH`, this will start R as a non interactive BATCH session. It runs a specified R file with some plotting functions.

Create an R file with some plot statements, the file plotR.R. The code in the file will instruct R to export the graph.

```
jpg(filename = "C:\\temp\\Rgraph.jpg")
x <- rnorm(100)
y <- rnorm(100)
par(mfrow=c(2,1))
plot(x,y)
hist(y)
dev.off()
```

Then in a SAS session, use the `call system` function to call an external program. In this case `Rcmd BATCH`.

```
%let myf = '"C:\Temp\plotR.R"';

data _null_;
  command='Rcmd BATCH '||&myf ;
  put command;
  call system(command);
run;
```

The same example but now using the `X` function in SAS.

```
%let myf = "C:\Temp\plotR.R";
%let  command=Rcmd BATCH  &myf ;
X &command;
```

9.3.2 Using SAS data sets and SAS ODS

The previous example used internal R data to create the plot. To create R graphs using SAS data sets you can

- export SAS data to a text file and import that in R,
- import the SAS data set in R directly.

The following example creates a small data set in SAS and exports it using `proc export`.

```
data testdata;
  input x y;
  datalines;
  1 3
  2 6
  3 8
```

```
run;

proc export data = testdata
  outfile = "C:\temp\sasdata.csv"
  REPLACE;
run;
```

Then in the R file plotR2.R we import the data and use the data to create a simple graph.

```
sasdata <- read.csv("C:\\temp\\sasdata.csv")
jpeg("C:\\temp\\Rgraph2.jpg")
plot(sasdata$x, sasdata$y)
dev.off()
```

Then in SAS we call `Rcmd BATCH` to run the above R file non interactively.

```
%let myf = '"C:\Temp\plotR2.R"';
data _null_;
  command='Rcmd BATCH '||&myf ;
  put command;
  call system(command);
run;
```

The SAS output delivery system (ODS) is a convenient system to create reports in HTML, PDF or other formats. The ODS takes output from SAS procedures and graphs, together with specific user settings it creates a certain report. The graphs don't have to be SAS graphs, they could be any graph.

Lets use the same dataset `testdata` as in the previous example. First run the SAS code that calls the R code that creates the graph.

```
%let myf = '"C:\Temp\plotR2.R"';
data _null_;
  command='Rcmd BATCH '||&myf ;
  put command;
  call system(command);
run;
```

When the graphs in R are created and are stored on disc, start the specifications of the SAS ODS:

```
ods html file = "sasreport.html";

title 'SAS output and R graphics';
title2 'a small example';

* Some SAS procedure that writes results in the report;
proc means data = Testdata;
run;

* export the SAS data and call R to create the plot;
proc export data = testdata
  outfile = "C:\temp\sasdata.csv"
  REPLACE;
run;

%let myf = '"C:\Temp\plotR2.R"';
data _null_;
  command='Rcmd BATCH '||&myf ;
  put command;
  call system(command);
run;

* insert additional html that inserts the graph that R created;
ODS html text = "<b> My Graph created in R </b>";
ODS html text = "<img src='c:\temp\Rgraph2.jpg' BORDER = '0'>";

ODS html close;
```

9.4 Defaults and preferences in R, Starting R,

9.4.1 Defaults and preferences

The function `options` is used to get and set a wide range of options in R. These options influence the way results are computed and displayed. The function `options` lists all options, the function `getOption` lists one specific option and `option(optionname = value)` sets a certain option.

```
options()
$add.smooth
[1] TRUE

$check.bounds
[1] FALSE
```

```
$chmhelp
[1] TRUE

$continue
[1] "+ "

$contrasts
        unordered           ordered
"contr.treatment"      "contr.poly"

$defaultPackages
[1] "datasets"  "utils"     "grDevices" "graphics"  "stats"     "methods"

$device
[1] "windows"

$digits
[1] 7
...
...
```

One example is the number of digits that is printed, by default the number is seven. This can be increased.

```
sqrt(2)
[1] 1.414214
options(digits=15)
sqrt(2)
[1] 1.41421356237310
```

See the help file of the function options for a complete list of all options.

9.4.2 Starting R

The online help describes precisely which initialization steps are carried out during the start up of R. Enter `?Startup` to see the help file.

If want to start R and set certain options or attach (load) certain packages automatically then this can be achieved by editing the file `Rprofile.site`. This file is located in the `etc` subdirectory of the R installation directory, so something like C:\Program Files\R-2.5.0\etc. The following file is just an example file.

```
# print extra digits
options(digits=10)

# papersize setting
options(papersize="a4")

# to prefer HTML help
options(htmlhelp=TRUE)

# adding libraries that should be attached
library(MASS)
library(lattice)
```

9.5 Creating an R package

9.5.1 A 'private' package

Once you start working with R, you will soon start creating your own functions. If these functions are used regularly (more than one or two times), it may be useful to collect these functions in a 'private' package. I.e. the functions are only used by you and some colleagues, and you want to have the functions in a separate library so that they don't pollute your workspace. In this case it may be enough to define the functions and save them to an R workspace image, a .RData file. This file is a binary file and can be attached to the R search path.

```
myf1 <- function(x)
{
  x^2
}

myf2 <- function(x)
{
  sin(x^2) + x
}

save(c("myf1", "myf2"), file = "C:\\MyRstuff\\AuxFunc.RData")
```

When a colleague needs these functions give him the binary file and let him attach it to his R session.

```
attach(C:\\MyRstuff\\AuxFunc.RData")
```

9.5.2 A 'real' R package

When you intend to make your package available for more people or even to the R community by putting it on CRAN you may want to create a 'real' R package. I.e. add documentation, create meaningful help files and make it easy to install and use the package. To achieve this, there are certain steps that you need to undertake. This section only gives you a kick in the right direction. For a complete description look at the R manual 'Writing R Extensions'.

Before you can create an R package you should install the following tools first, the tools can be found on http://www.murdoch-sutherland.com/Rtools.

- Perl, a scripting language.
- Rtools.exe, a collection of command line tools and compilers.
- Microsoft HTML help workshop, to create help files.
- MikTex, a LaTex and pdftex package.

When the tools are installed, your PATH variable should be edited, so that commands can be found. You need to be careful in specifying the order of the directories. For example, if you also have the MAKE utility of Borland then make sure that your system finds the R MAKE first when building R packages. Depending on the installation directories, your path may look like:

```
PATH = C:\Rtools\bin;
       C:\perl\bin;
       C:\Rtools\MinGW\bin;
       C:\Program Files\HTML Help Workshop;
       C:\Program Files\R\R-2.5.0\bin;
       C:\texmf\miktex\bin;
       <others>
```

A good starting point to create an R package is the function `package.skeleton`. We create a package 'Lissajous' with two functions that plot Lissjous figures to demonstrate the necessary steps.

Define the R functions

First create a script file that defines the functions that you want to package. In our case we have the following function definitions.

```
LissajousPlot <- function(nsteps, a,b)
{
  t <- seq(0,2*pi, l = nsteps)
  x <- sin(a*t)
  y <- cos(b*t)
  plot(x,y,type="l")
}

LissajousPlot2 <- function(nsteps, tend, a,b,c)
{
  t <- seq(0, tend, l= nsteps)
  y = c*sin(a*t)*(1 + sin(b*t))
  x = c*cos(a*t)*(1 + sin(b*t))
  plot(x,y,type="l")
}
```

Test the functions, make sure the functions produce the results you expect.

Run the function `package.skeleton`

The function `package.skeleton` creates the necessary files and sub directories that are needed to build the R package. It allows the user to specify which objects will be placed in the package. Specify a name and location for the package:

```
package.loc = "C:\\RPackages"
package.skeleton("Lissajous", path = package.loc, force=T)
Creating directories ...
Creating DESCRIPTION ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in 'C:\RPackages/Lissajous/Read-and-delete-me'.
```

The above call will put all objects in the current workspace in the package, use the `list` argument to specify only the objects that you want to put in the package.

```
package.skeleton("Lissajous",
   path = package.loc,
   list = c("LissajousPlot", "LissajousPlot2"),
   force = T
)
```

If `force = T` then R will overwrite an existing directory. Note that previously edited DESCRIPTION and Rd help files are overwritten!! If the function has finished, the directory 'Lissajous' and some subdirectories are created.

Edit and create some files

The DESCRIPTION file, is a basic description of the package. R has created a skeleton that the user can edit. We use the following file.

```
Package: Lissajous
Type: Package
Title: Create Lissajous figures
Version: 1.0
Date: 2007-05-09
Author: Longhow Lam
Maintainer: Longhow Lam <longhow.lam@businessdecision.com>
Description: Create Lissajous figures
License: no restrictions
```

This information appears for example when you display the general help of a package.

```
help(package="Lissajous")
```

The INDEX file is not created, it is an optional file that lists the interesting objects of the package. We use the file:

```
LissajousPlot          Plot a Lissajous figure
LissajousPlot2         Plot another Lissajous figure
```

Create help and documentation

The function `package.skeleton` has also created initial R help files for each function, the *.Rd files in the `man` subdirectory. R help files need to be written in 'R documentation' format. A markup language that closely resembles LaTex. The initial files should be edited to provide meaningful help. Fortunately, the initial Rd files created by R provide a good starting point. Open the files and modify them.

When the package is build, these documentation files are compiled to html and Windows help files. Each function should have a help file, it is the help that will be displayed when a user uses the `help` function.

```
help(LissajousPlot2)
```

Build the package

Now the necessary steps are completed, the package can be build. Open a DOS box, go to the directory that contains the 'Lissajous' directory and run the command:

```
> Rcmd build --binary Lissajous
```

When the build is successful, you should see the zip file: `Lissajous_1.0.zip`.

Install and use the package

In the RGui window go to the menu 'Packages' and select 'Install package(s) from local zip files...'. Then select the `Lissajous_1.0.zip` file, R will install the package. To use the package, it should be attached to your current R session.

```
library(Lissajous)
help(Lissajous)
par(mfrow=c(2,2))
LissajousPlot(300,2,5)
LissajousPlot(300,14,4)
LissajousPlot2(300,10,2,7,5)
LissajousPlot2(300,10,100,25,6)
```

9.6 Calling R from Java

The package 'rJava' can be used to call java code within R, which is comparable with the technology described in section 6.3. The other way around is also possible, calling R from java code. This is implemented in the JRI package which will also be installed if you install the rJava package. This could become useful when you want to extend your java programs with the numerical power of R, or build java GUI's around R. This section demonstrates the latter. We will make use of the NetBeans IDE. This is powerful yet easy to use environment for creating java (GUI) applications, it is freely available from www.netbeans.org.

So to replicate the example in this section download the following software:

- Java development Kit (JDK)
- The NetBeans IDE
- The R package 'rJava'

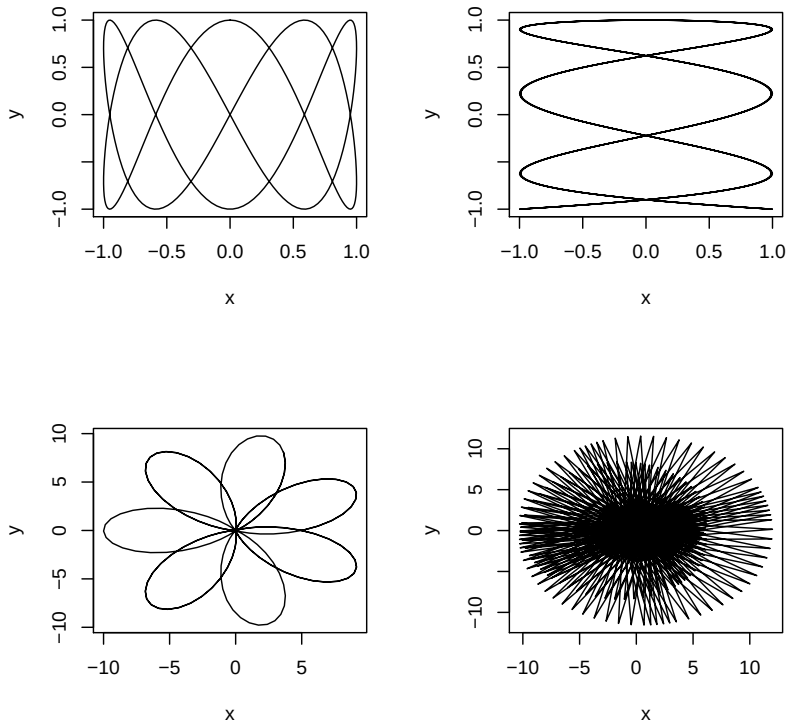

Figure 9.2: Some Lissajous plots

The next figure shows a small application that allows the user to import a text file, create explorative plots and fit a regression model. The NetBeans project and java code files are available from the website of this document. The code is not that difficult. Most of the work is done in the JRI package which contains an 'REngine' object that you can embed in your java code.

A brief description of the java gui. A global REngine object `re` is defined and created in the java code.

```
Rengine re = new Rengine(args, false, new TextConsole());
```

Throughout the java program the object `re` can be used to evaluate R expressions. For example, if the 'Import Data' button is clicked an import file dialog appears that will return a `filename`, then the following java code is called:

```
String evalstr = "infile <- " + "\"" + filename + "\"";
re.eval(evalstr);
String impstr = "indata = read.csv(infile)";
re.eval(impstr);
```

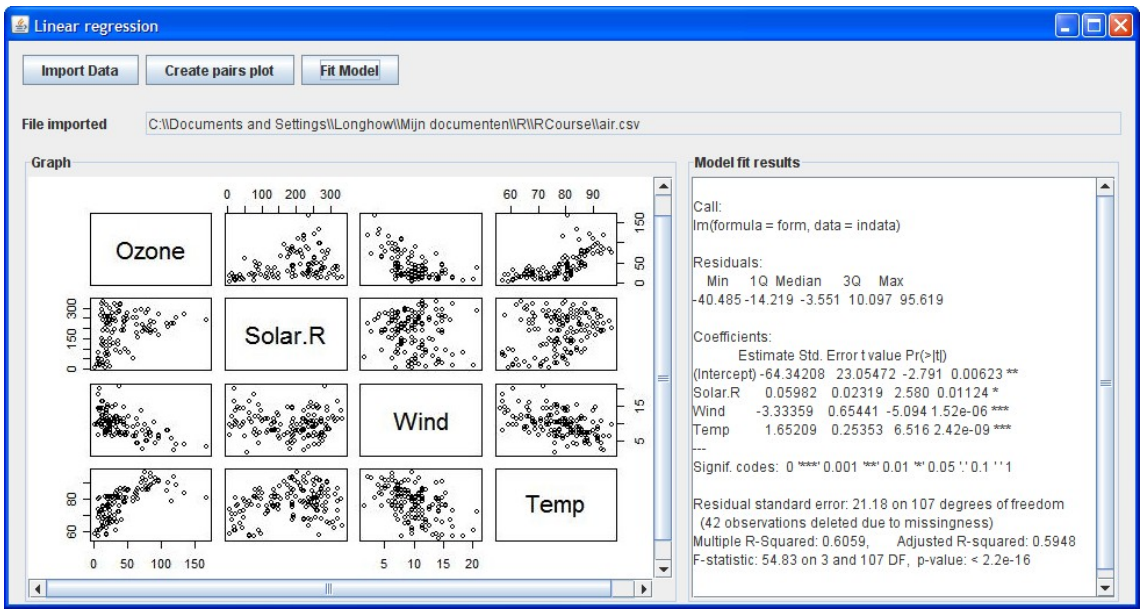

Figure 9.3: A small java gui that can call R functions.

This will cause R to call the `read.csv` function and create an R object infile. Then if the user click on the 'Create pairs plot' button, the user can select the variables that will be plotted in a pairs plot. The java program will run:

```
String filename;
filename = "C:/Temp/Test.jpg";
String evalstr = "plotfile  <- " + "\"" + filename + "\"";
re.eval(evalstr);
re.eval("jpeg(plotfile, width=550, height=370)");
re.eval("pairs(indata[colsel])");
re.eval("dev.off()");
```

So the REngine object `re` is used to evaluate the `pairs` function and store the result in a jpeg file. This jpeg file is picked up by the java gui (in a JLabel object), so that it is visible. Then when the user clicks on 'Fit Model', a dialog will appear where the user selects the response and the regression variables. The R engine is called to fit the linear regression model. The output is displayed in the results window.

9.7 Creating fancy output and reports

The R system contains several functions and methods that facilitate the user to create fancy output and reports. First a short overview of these methods, then some examples.

- Instead of the normal output to screen, the function `sink` redirects the output of R to a connection or external file.
- The package 'xtable' contains functions to transform an R object to an xtable object which can then be printed to HTML or LaTeX.
- The package 'R2HTML' contains functions to create HTML code from R objects.
- The functions `jpeg` and `pdf` (see section 7.2.4) export R graphics to external files in jpeg and pdf format. These files can then be included in a web page or document.
- Sweave is a tool that can process a document with chunks of R code, see [15]. It parses the document, evaluates the chunks of R code and puts the resulting output (text and graphs) back in the document in such a way that the resulting document is in its native format. The formats that are implemented are LaTeX, HTML and ODF (Open Document Format).

9.7.1 A simple LaTeX-table

In a monthly report that is created in LaTeX, the output of a linear regression in R is needed.

```
## load the xtable package
library(xtable)

## specify the file that will contain the regression output in Latex format
mydir = "C:\\Documents and Settings\\Longhow\\Mijn Documenten\\R\\RCourse\\"
outfile <- paste(mydir,"carslm.tex",sep="")

## Fit a linear regression
lm.out <- lm(Price ~ Mileage + Weight + HP, data = cars)

## transform the regression output object to an xtable object
## add a label so that the table can be referenced in Latex
lm.out.latex <- xtable(
     lm.out,
     caption = "Regression output",
     label = "tab001",
     type = "latex"
)

## sink the xtable object to the latex file.
sink(outfile)
print(lm.out.latex)

## redirect output to normal screen
```

```
sink()
```

Once the latex file has been created it can be imported in the the LaTeXreport with the `input` command in latex. See Table 9.1.

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	4236.9773	7409.1846	0.57	0.5697
Mileage	−161.5201	146.5253	−1.10	0.2750
Weight	2.7349	1.6323	1.68	0.0994
HP	36.0914	18.5871	1.94	0.0572

Table 9.1: Regression output

9.7.2 An simple HTML report

A small demonstration of Sweave. Every month you need to publish a report that includes some summary statistics and a graph on your local intranet site. Create a file, say `datareport`. Treat this file as a 'normal' HTML file.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
  <title>data report</title>
</head>
<body>
<h2>Monthly summary of input data</h2>
<br>
Data summary  of  sales data from this month

<<echo = FALSE>>=
out <- var(cars[c("Price", "Weight", "Mileage")])
out <- xtable(out,caption="Correlation of this months price data")
print(out,type="HTML")
@

A graph of the data

<<fig=TRUE, echo = FALSE>>=
pairs(cars[c("Price", "Weight", "Mileage")])
@
</body>
</html>
```

The chunks of R code start with `<<` *some options* `>>=` and end with an `@`. There are a few options you can set.

- `echo = FALSE`, the R statements in the chunk are not put in the output. Useful when some R statements need to run, for example importing or manipulating data, but need not to be visible in the final report.
- `results=hide`, will hide any output. However, it will generate the R statements in the final document when the `echo` option is not set to `FALSE`.
- `fig=TRUE`, create a figure in the report when the R code contains plotting commands.

Save the file when you are ready and use `Sweave` in R to process this file.

```
library(R2HTML)
mydir = "C:\\Documents and Settings\\Longhow\\Mijn Documenten\\R\\RCourse\\"
myfile <- paste(mydir,"data_report",sep="")
Sweave(myfile, driver=RweaveHTML)
Writing to file data_report.html
Processing code chunks ...
 1 : term Robj
 2 : term Robj png
file  data_report.html is completed
```

The result is an ordinary HTML file that can be opened by a web browser.

Bibliography

[1] Longhow Lam, *A guide to Eclipse and the R plug-in StatET*. www.splusbook.com, 2007.

[2] Diethelm Würtz, "S4 'timedate' and 'timeseries' classes for R," *Journal of Statistical Software*.

[3] Robert Gentleman and Ross Ihaka, "Lexical scope and statistical computing," *Journal of Computational and Graphical Statistics*, vol. 9, p. 491, 2000.

[4] W. N. Venables and B. D. Ripley, *S Programming*. Springer, 2000.

[5] D. Samperi, "Rcpp: R/C++ interface classes, using c++ libraries from R," 2006.

[6] P. Murrell, *R Graphics*. Chapman & Hall, 2005.

[7] Hadley Wickham, *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2009.

[8] Leland Wilkinson, *The Grammar of Graphics*. Springer, 2005.

[9] W. N. Venables and B. D. Ripley, *Modern Applied Statistics with S*. Springer, September 2003.

[10] J. Maindonald and J. Braun, *Data Analysis and Graphics Using R: An Example-based Approach*. Cambridge University Press, 2007.

[11] T. Hastie , R. Tibshirani , J. H. Friedman, *The Elements of Statistical Learning* . Springer, 2001.

[12] M. Prins and P. Veugelers, "The european seroconverter study and the tricontinental seroconverter study. comparison of progression and non-progression in injecting drug users with documented dates of hiv-1 seroconversion.," *AIDS*, vol. 11, p. 621, 1997.

[13] T. M. Therneau and P. M. Grambsch, *Modeling Survival Data: Extending the Cox Model*. Springer, 2000.

[14] Douglas M. Bates, Donald G. Watts, *Nonlinear regression analysis and its applications*. Wiley-Interscience, 2007.

[15] Friedrich Leisch, "Sweave user manual," 2006.

Index

Accelerated failure time model, 164
`aggregate`, 60
`apply`, 86
area under ROC, 157
`array`, 35
arrows, 119
`as.difftime`, 26
`as.list`, 41
attributes, 62
axes, 121
`axis`, 121

bar plot, 107
binning, 162
`break`, 80
`browser`, 83
`by`, 90

C, 92
`c`, 28
calls, 189
cars, example data, 54
`cbind`, 56
character, 22
character manipulation, 63
chol, 34
Churn analysis, 164
coarse classification, 162
color palette, 117
color symbols, 117
compiled code, 92
compilers, 95
complex, 21
concordant, 158
conditioning plots, 123, 130
conflicting objects, 15
`conflicts`, 15
control flow, 77
Cox proportional hazard model, 164
csv files, 42
cumulative sum, 49
`curve`, 104
`cut`, 68

data frames, 35
databases, 45
`debug`, 82
debugging, 80
delimited files, 42
`deriv`, 178
difftime, 26
`dim`, 32
double, 19
`duplicated`, 50

eclipse, 16
`eval`, 189
Excel files, 44
expressions, 189

Facetting, 133
factor, 22
factor variables, 152
`FALSE`, 21
figure region, 114
font, 117
`for`, 79
formula objects, 141
Fortran, 92
free variables, 75

`glm`, 155
Graphical Devices, 110

`grep`, 65
`gsub`, 67

`head`, 55
help, 11
`help`, 19
HTML, 204

`if`, 77
ill-conditioned models, 174
import data, 42
integer, 20
`is.infinite`, 27
`is.na`, 27
`is.nan`, 27

join, 59
jpeg, 112

Kendall's tau-a, 158

language objects, 189
`lapply`, 87
Latex, 204
`layout`, 115
`layout.show`, 115
lazy evaluation, 76
legends, 120
`length`, 49
level, 23
`levels`, 23
lexical scope, 75
line type, 117
line width, 117
Linear regression, 142
lines, 119
lists, 38
local variables, 73
logical, 21
logistic regression, 154
loops, 77
low level plot functions, 119
`lrm`, 159

margins, 114
masked objects, 15
mathematical expressions in graphs, 120
Mathematical operators, 29
`matrix`, 31
`merge`, 59
model diagnostics, 146
`mtext`, 120
multicollinearity, 149
multiple plots per page, 115

`NA`, 27
`NaN`, 27
`nchar`, 64
Non linear regression, 170
`NULL`, 28

object oriented programming, 180
ODBC, 45
`option`, 196
`order`, 50
ordered factors, 24

package, 13
package creation, 198
`paste`, 64
pie plot, 107
`plot`, 103
plot region, 114
polynomial contrast, 152
POSIXct, 25
POSIXlt, 25
predictive ability, 158
preferences, 196
probability distributions, 139
`proc.time`, 85

ragged arrays, 89
random sample, 139
`rbind`, 57
`rbind.fill`, 58
`read.table`, 42
Receiver Operator curve, 157
Recycling, 29
`regexpr`, 65
regular expressions, 65
`rep`, 31

`repeat`, 80
replacing characters, 67
reports, 204
`reshape`, 61
reshape package, 58
`return`, 75
`round`, 29

S3 classes, 180
S4 classes, 180
sample, 139
`sapply`, 87
`scan`, 44
scoping rules, 75
`search`, 13
search path, 13
segments, 119
sensitivity marix, 177
sequences, 30
`sessionInfo`, 14
singular value decomposition, 151, 177
solve, 34
Somer's D, 158
sort, 50
`stack`, 61
stacking data frames, 57
start up of R, 197
statistical summary functions, 135
stop, 81
`str`, 41
`strptime`, 25
`strsplit`, 68
structure, 41
`sub`, 67
subset, 56
`subset`, 56
`substring`, 64
`survreg`, 169
svd, 34
Sweave, 204
`switch`, 78
symbols, 117
`Sys.time`, 27
`system.time`, 85

`tail`, 55
`tapply`, 89
`terms`, 143
text files, 42
time-series, 37
Tinn-R, 17
titles, 103
`traceback`, 80
transpose, 34
treatment contrast, 152
tree models, 160
Trellis plots, 123
`TRUE`, 21
`tsp`, 38
typeof, 19

`unique`, 50

variance inflation factors, 151
vector, 28
vector subscripts, 47
`vif`, 151

warning, 81
`while`, 79
working directory, 12
workspace image, 12