Basic R for Finance

Diethelm Würtz
Longhow Lam
Andrew Ellis
Yohan Chalabi

Rmetrics Association & Finance Online

R/Rmetrics eBook Series

R/Rmetrics eBooks is a series of electronic books and user guides aimed at students and practitioner who use R/Rmetrics to analyze financial markets.

A Discussion of Time Series Objects for R in Finance (2009)
Diethelm Würtz, Yohan Chalabi, Andrew Ellis

Portfolio Optimization with R/Rmetrics (2010),
Diethelm Würtz, William Chen, Yohan Chalabi, Andrew Ellis

Basic R for Finance (2010),
Diethelm Würtz, Yohan Chalabi, Longhow Lam, Andrew Ellis
Early Bird Edition

Financial Market Data for R/Rmetrics (2010)
Diethelm Würtz, Andrew Ellis, Yohan Chalabi
Early Bird Edition

Indian Financial Market Data for R/Rmetrics (2010)
Diethelm Würtz, Mahendra Mehta, Andrew Ellis, Yohan Chalabi

R/Rmetrics Workshop Singapore 2010 (2010)
Diethelm Würtz, Mahendra Mehta, David Scott, Juri Hinz

BASIC R FOR FINANCE

DIETHELM WÜRTZ
LONGHOW LAM
ANDREW ELLIS
YOHAN CHALABI

RMETRICS ASSOCIATION & FINANCE ONLINE

Series Editors:
PD Dr. Diethelm Würtz
Institute of Theoretical Physics and
Curriculum for Computational Science
Swiss Federal Institute of Technology
Hönggerberg, HIT K 32.2
8093 Zurich

Contact Address:
Rmetrics Association
Weinbergstrasse 41
8006 Zurich
info@rmetrics.org

Authors:
Yohan Chalabi, ETH Zurich
Andrew Ellis, Rmetrics Association
Longhow Lam, ABN AMRO
Diethelm Würtz, ETH Zurich

Dr. Martin Hanf
Finance Online GmbH
Weinbergstrasse 41
8006 Zurich

Publisher:
Finance Online GmbH
Swiss Information Technologies
Weinbergstrasse 41
8006 Zurich

Cover Design & Photography:
Chris Eckert
Leaf dtp production

ISBN:
eISBN:
DOI:

DEDICATION

This book is dedicated to all those who
have helped make Rmetrics what it is today:
The leading open source software environment in
computational finance and financial engineering.

PREFACE

R und Rmetrics are the talk of the town. The statistical software package R is one of the most promising tools for rapid prototyping of financial applications. The userR conferences and Rmetrics Meielisalp workshops reflect the growing interest in R und Rmetrics.

Have ever thought of giving R a try, using one of the many packages, or even writing your own functions and learning the programming language? If only the initial learning curve weren't so steep. This is where "Basic R for Finance" can help you: You will learn the basics of programming in R, and how to implement your models and applications. You will not only create graphics, you will also learn how to customize them them to suit your needs. You will learn how to write your own programs, how to write efficient and optimized code. Furthermore, you will be assisted by a multitude of very detailed case studies, ranging from computing skewness and kurtosis statistics to designing and optimizing portfolios of assets.

This book is divided into two several thematically distinct parts: the first four parts give an introduction to R, and focus on the following topics: computions, programming, plotting and statistics and inference. Parts five, six, svem and eight contain a collection of case studies from topics such as utility functions, asset management, option valuation, and portfolio design.

We hope you enjoy this book!

Diethelm Würtz
Zurich, July 2010

Contents

List of Figures

List of Tables

PART I

COMPUTATIONS

CHAPTER 1

DATA STRUCTURES

Data structures describe various way of coherently organizing data. The most common data structures in R are:

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

1.1 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, for example all doubles or all logical, as is called an atomic object. 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] 10.00  5.00  3.00  6.00  0.55 10.00  5.00  3.00  6.00 10.00  5.00  3.00
[13]  6.00
```

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)
```

```
 [1] 10.00  5.00  3.00  6.00  0.55 10.00  5.00  3.00  6.00 10.00  5.00  3.00
[13]  6.00
```

Mathematical operators

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`.

LISTING 1.1: SOME MATHEMATICAL FUNCTIONS THAT CAN BE APPLIED TO VECTORS.

```
Function:
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
```

```
> 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] 2.3026 1.6094 1.0986 1.7918
```

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 to recall a vector of length one.

```
> sqrt(x) + 2
[1] 5.1623 4.2361 3.7321 4.4495
```

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
> z
[1]  1  4  9 16  5 12
```

Generating vectors with the (:) column operator

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 sequence function `seq()`

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 themselves.

```
> seq(as.POSIXct("2003-04-23"), by = "month", length = 12)
 [1] "2003-04-23 CEST" "2003-05-23 CEST" "2003-06-23 CEST" "2003-07-23 CEST"
 [5] "2003-08-23 CEST" "2003-09-23 CEST" "2003-10-23 CEST" "2003-11-23 CET"
 [9] "2003-12-23 CET"  "2004-01-23 CET"  "2004-02-23 CET"  "2004-03-23 CET"
> iso.tS = seq(ISOdate(1910, 1, 1), ISOdate(1999, 1, 1), "years")
> head(iso.tS, 12)
 [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"
 [5] "1914-01-01 12:00:00 GMT" "1915-01-01 12:00:00 GMT"
 [7] "1916-01-01 12:00:00 GMT" "1917-01-01 12:00:00 GMT"
 [9] "1918-01-01 12:00:00 GMT" "1919-01-01 12:00:00 GMT"
[11] "1920-01-01 12:00:00 GMT" "1921-01-01 12:00:00 GMT"
```

The repeat function `rep()`

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)
> y <- runif(10, 4, 7)
```

1.2 MATRICES

A matrix can be regarded as a vector with a special dimension attribute. As with vectors, all the elements of a matrix must be of the same data type. A matrix can be generated in several ways.

Converting vectors to matrices with the `dim()` function

Use the function `dim()` to convert a vector into a matrix. It does internally add the special attribute "dim" to the vector.

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

Generate matrices with the `matrix()` function

Alternatively use the function `matrix()` to generate a matrix object.

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

Note by default the matrix is filled by column as in the previous example. To fill the matrix by row specify `byrow = TRUE` as argument in the matrix function.

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

Binding vectors column and row wise

Use the function `cbind()` to create a matrix by binding two or more vectors as column vectors.

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

The function `rbind()` is used to create a matrix by binding two or more vectors as row vectors.

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

Calculations on matrices

Since a matrix is a vectors with a special attribute, all the mathematical functions that apply to vectors also apply to matrices and are applied on each matrix element.

```
> x * x^2
```

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

```
> max(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.

Matrix multiplication

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

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

The transposed matrix

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

```
> x %*% t(x)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]  774  820  866  912  958 1004 1050
[2,]  820  870  920  970 1020 1070 1120
[3,]  866  920  974 1028 1082 1136 1190
[4,]  912  970 1028 1086 1144 1202 1260
[5,]  958 1020 1082 1144 1206 1268 1330
[6,] 1004 1070 1136 1202 1268 1334 1400
[7,] 1050 1120 1190 1260 1330 1400 1470
```

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

LISTING 1.2: SOME FUNCTIONS THAT CAN BE APPLIED ON MATRICES.

```
Function:
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
```

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.

1.3 ARRAYS

Arrays are vectors with a dimension attribute specifying more than two dimensions. 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.

```
> x <- 1:8
> dim(x) <- c(2, 2, 2)
> x
, , 1

     [,1] [,2]
```

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

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8


> dim(iris3)
[1] 50  4  3
```

All basic arithmetic operations which apply to vectors 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
```

1.4 DATA FRAMES

Data frames can be regarded as lists with element of the same length that are represented in a two dimensional object. 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 Longley's Economic Data set.

```
> data(longley)
> longley
```

```
     GNP.deflator    GNP Unemployed Armed.Forces Population Year Employed
1947         83.0 234.29      235.6        159.0     107.61 1947   60.323
1948         88.5 259.43      232.5        145.6     108.63 1948   61.122
1949         88.2 258.05      368.2        161.6     109.77 1949   60.171
1950         89.5 284.60      335.1        165.0     110.93 1950   61.187
1951         96.2 328.98      209.9        309.9     112.08 1951   63.221
1952         98.1 347.00      193.2        359.4     113.27 1952   63.639
1953         99.0 365.38      187.0        354.7     115.09 1953   64.989
1954        100.0 363.11      357.8        335.0     116.22 1954   63.761
1955        101.2 397.47      290.4        304.8     117.39 1955   66.019
1956        104.6 419.18      282.2        285.7     118.73 1956   67.857
1957        108.4 442.77      293.6        279.8     120.44 1957   68.169
1958        110.8 444.55      468.1        263.7     121.95 1958   66.513
1959        112.6 482.70      381.3        255.2     123.37 1959   68.655
1960        114.2 502.60      393.1        251.4     125.37 1960   69.564
1961        115.7 518.17      480.6        257.2     127.85 1961   69.331
1962        116.9 554.89      400.7        282.7     130.08 1962   70.551
```

The data set list from 1947 to 1962 U.S. economic data including the GNP implicit price deflator, Gross national Product GNP, number of unemployed, number of people in the armed forces, the year, and the number of people employed.

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 `rownames()`. The result is a character vector containing the names.

```
> rownames(longley)

 [1] "1947" "1948" "1949" "1950" "1951" "1952" "1953" "1954" "1955" "1956"
[11] "1957" "1958" "1959" "1960" "1961" "1962"

> names(longley)

[1] "GNP.deflator" "GNP"          "Unemployed"   "Armed.Forces" "Population"
[6] "Year"         "Employed"
```

Creating data frames

You can create data frames in several ways, by importing a data frame from a data file for example, or by using the function `data.frame()`. This function can be used to create new data frames or to convert other objects into data frames.
An examples how to create a `data.frame()` from scratch:

```
> myLogical <- sample(c(TRUE, FALSE), size = 10, replace = TRUE)
> myNumeric <- rnorm(10)
> myCharacter <- sample(c("AA", "A", "B", "BB"), size = 10, replace = TRUE)
> myDataFrame <- data.frame(myLogical, myNumeric, myCharacter)
> myDataFrame
   myLogical myNumeric myCharacter
1       TRUE   2.05236           A
2       TRUE  -0.63297           B
3       TRUE   0.59791           A
4       TRUE   1.22769           B
5      FALSE   0.97853           B
6       TRUE   0.32813          AA
7      FALSE   0.14386          BB
8      FALSE  -0.17417          AA
9      FALSE  -0.18186          AA
10     FALSE   1.13973          BB
```

1.5 TIME SERIES

In R a time series object can be created with the function `ts()` which returns an object of class `"ts"`.

```
> args(ts)
function (data = NA, start = 1, end = numeric(0), frequency = 1,
    deltat = 1, ts.eps = getOption("ts.eps"), class = if (nseries >
        1) c("mts", "ts") else "ts", names = if (!is.null(dimnames(data))) colnames(data) else paste("Series",
        seq(nseries)))
NULL
```

LISTING 1.3: ARGUMENTS OF THE FUNCTION TS.

```
Arguments:
data        numeric vector or matrix of the observed values
start       time of the first observation
end         time of the last observation
frequency   number of observations per unit of time.
deltat      fraction of sampling period between observations
ts.eps      time series comparison tolerance
class       class to be given to the result
names       character vector of names for multiple time series
```

The function `ts()` combines two components, (i) the data, a vector or matrix of numeric values, and (ii) the time stamps of the data. Note the time stamps are always equispaced points in time. In this case we say the function `ts()` generates regular tim series.
Here is an example from R's `UKgas` data file

```
> UKgas
```

```
       Qtr1   Qtr2   Qtr3   Qtr4
1960  160.1  129.7   84.8  120.1
1961  160.1  124.9   84.8  116.9
1962  169.7  140.9   89.7  123.3
1963  187.3  144.1   92.9  120.1
1964  176.1  147.3   89.7  123.3
1965  185.7  155.3   99.3  131.3
1966  200.1  161.7  102.5  136.1
1967  204.9  176.1  112.1  140.9
1968  227.3  195.3  115.3  142.5
1969  244.9  214.5  118.5  153.7
1970  244.9  216.1  188.9  142.5
1971  301.0  196.9  136.1  267.3
1972  317.0  230.5  152.1  336.2
1973  371.4  240.1  158.5  355.4
1974  449.9  286.6  179.3  403.4
1975  491.5  321.8  177.7  409.8
1976  593.9  329.8  176.1  483.5
1977  584.3  395.4  187.3  485.1
1978  669.2  421.0  216.1  509.1
1979  827.7  467.5  209.7  542.7
1980  840.5  414.6  217.7  670.8
1981  848.5  437.0  209.7  701.2
1982  925.3  443.4  214.5  683.6
1983  917.3  515.5  224.1  694.8
1984  989.4  477.1  233.7  730.0
1985 1087.0  534.7  281.8  787.6
1986 1163.9  613.1  347.4  782.8
```

```
> class(UKgas)
```

```
[1] "ts"
```

The following examples show how to create objects of class `"ts"` from scratch.

Create a monthly series

Create a time series of random normal deviates starting from January 1987 with 100 monthly intervals

```
> ts(data = round(rnorm(100), 2), start = c(1987), freq = 12)
```

```
       Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov   Dec
1987  1.64  0.65  0.95 -0.05  1.66  0.06  1.28 -1.40  0.60 -0.31 -0.09 -0.36
1988 -0.23 -0.47  2.52  0.01  1.87 -0.21  1.77 -0.43  0.80  0.84 -0.67  0.60
1989 -0.43  1.59  1.71  0.86  0.55 -0.42  0.33  1.04 -1.56 -0.61 -0.14 -1.24
1990 -0.67 -0.11  0.98 -1.68 -0.62  0.51  2.07  0.41 -0.66  0.23  0.59 -0.91
1991  1.37  0.98  1.42 -1.47  0.25  1.59 -2.09  0.75 -0.22 -0.09 -0.58  0.21
1992 -0.03 -0.43  0.08  0.64  1.09  0.59 -0.56 -0.90  0.67 -1.64  0.68 -2.15
1993  0.45  0.95 -0.17  0.34 -1.22 -0.28 -1.50  0.85  1.15 -0.86  1.05 -0.70
1994  0.23 -1.52 -1.02 -0.25 -2.92  1.90  0.26  0.37  0.19  0.59  1.16  2.22
1995 -0.19 -0.06  0.48 -0.91
```

```
> class(ts)
```

```
[1] "function"
```

Create a multivariate time series

Now create a bivariate time series starting from April 1987 with 12 monthly intervals

```
> ts(data = matrix(rnorm(24), ncol = 2), start = c(1987, 4), freq = 12)
          Series 1    Series 2
Apr 1987 -1.166093 -0.38528737
May 1987 -0.550545 -0.19529599
Jun 1987  0.027019  0.10292740
Jul 1987  0.515447 -0.00083421
Aug 1987 -1.548343  2.13707625
Sep 1987 -0.037109 -0.10832004
Oct 1987  1.131966 -0.84597308
Nov 1987  0.223646  1.39331542
Dec 1987 -0.667762 -0.46451751
Jan 1988 -1.343814  0.59557268
Feb 1988 -0.345812 -0.86521684
Mar 1988 -0.860946 -0.32484920
```

```
> class(ts)
```

```
[1] "function"
```

The function `tsp()`

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

```
> tsp(ts(rnorm(48), start = 1987, freq = 4))
```

```
[1] 1987.0 1998.8    4.0
```

1.6 LISTS

A list is a vector. However, the contents of a list can be an object of any type and structure. It is a non-atomic object. Consequently, a list can contain another list and 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. The function `list()` has only one argument, the `...` argument.

```
> args(list)
```

```
function (...)
NULL
```

LISTING 1.4: ARGUMENTS OF THE FUNCTION LIST

```
Argument:
...             objects, possibly named
```

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

```
> x <- 1:5
> y <- x + rnorm(5, mean = 0, sd = 0.25)
> fit <- lsfit(x, y)
> fit

$coefficients
Intercept         X
 -0.33954   1.15505

$residuals
[1] -0.040323  0.310736 -0.130022 -0.510874  0.370482

$intercept
[1] TRUE

$qr
$qt
[1] -6.989082  3.652594 -0.061204 -0.350866  0.621679

$qr
     Intercept        X
[1,]  -2.23607 -6.70820
[2,]   0.44721  3.16228
[3,]   0.44721 -0.19544
[4,]   0.44721 -0.51167
[5,]   0.44721 -0.82790

$qraux
[1] 1.4472 1.1208

$rank
[1] 2

$pivot
[1] 1 2

$tol
[1] 1e-07

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

In this example the output value of `lsfit(x, y)` is assigned to object `fit`. 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:

- element number: `z[1]` means the first element of the list `z`. It therefore returns a `list` object.
- component number: `z[[1]]` means the content of the first component of `z` (use double square brackets!).
- component name: `z$name` indicates the component of `z` with name `name`. It retrieves the content of the element by its name.

To identify the component name the first few characters will do, for example, you can use `z$r` instead of `z$residuals`. But note that using incomplete component names are highly discourage in any R code.

```
> test <- fit$r
> test
[1] -0.040323  0.310736 -0.130022 -0.510874  0.370482
> fit$r[4]
[1] -0.51087
```

Creating lists from scratch

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(TRUE, TRUE, FALSE, FALSE, TRUE)
> myList <- list(numbers = x1, wrong = x2)
> myList
$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 `myList`.

```
> myList[[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(myList)
[1] "numbers" "wrong"
```

```
> names(myList) <- c("lots", "valid")
> names(myList)
[1] "lots"  "valid"
```

To add extra components to a list proceed as follows:

```
> myList[[3]] <- 1:50
> myList$test <- "hello"
> myList
$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.

```
> myList[1]
$lots
[1] 1 2 3 4 5
> myList[[1]]
[1] 1 2 3 4 5
```

Note when single square brackets are used, the component is returned as list because it extracts the first element of a list which is a list, whereas double square brackets return the component itself of the element.

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

[[4]]
[1] 4
```

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

[[6]]
[1] 6
```

1.7 PRINTING THE OBJECT STRUCTURE

A handy function is the structur function `str()`, it displays the internal structure of an R object. The function can be used to see a short summary of an object.
Show the structure for Longley's economic data set

```
> str(longley)
'data.frame': 16 obs. of  7 variables:
 $ GNP.deflator: num  83 88.5 88.2 89.5 96.2 ...
 $ GNP         : num  234 259 258 285 329 ...
 $ Unemployed  : num  236 232 368 335 210 ...
 $ Armed.Forces: num  159 146 162 165 310 ...
 $ Population  : num  108 109 110 111 112 ...
 $ Year        : int  1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 ...
 $ Employed    : num  60.3 61.1 60.2 61.2 63.2 ...
```

Show the structure for the quarterly UK gas price series

```
> str(UKgas)
 Time-Series [1:108] from 1960 to 1987: 160.1 129.7 84.8 120.1 160.1 ...
```

CHAPTER 2

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.

2.1 MANIPULATING VECTORS

Subsetting by positive natural numbers

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
```

To extract the first three numbers, type

```
> x[1:3]
[1] 6 7 2
```

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

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

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 with the same value, type

```
> 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)
Warning message:
In x[1:3] <- c(1, 2) :
  number of items to replace is not a multiple of replacement length
```

Because the replacement vector is shorter than the vector to be replaced, its elements are recycled; the third element is just the first element again.

Subsetting by a logical vector

Subsetting a vector by logical values results in a vector with only those elements of x of which the logical vector has an element TRUE.

```
> x <- c(10, 4, 6, 7, 8)
> index <- x > 9
> index
[1]  TRUE FALSE FALSE FALSE FALSE
> x[index]
[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. It will be recycled.
The recycling behavior can be handy in situation were one wants, for example, extract elements of a vector at even positions. This could be achieved by using a logical vector of length two.

```
> x[c(FALSE, TRUE)]
[1] 4 7
```

Subsetting by negative natural numbers

If you use negative natural numbers in the indexing vector, all elements of x are selected except those that are in the index.

```
> x <- c(1, 2, 3, 6)
> x[-(1:2)]
[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
```

The length of a vector: `length()`

The function `length()` returns the number of elements in a vector

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

Summary functions

There are several summary functions for working with vectors: `max()`, `min()`, `range()`, `prod()`, `sum()`, `any()`, `all()`. These functions are all so-called generic functions. Type

```
help(S3groupGeneric)
```

to find out more about these functions.
These functions allow us to calculate (for numerical values) the following statistics: the sum of all elements of the vector, the product, and the largest and smallest values of the vector. The range returns the minimum and maximum values of the vector together.

```
> x <- 1:10
> c(sum = sum(x), prod = prod(x), min = min(x), max = max(x))
```

```
    sum    prod     min     max
     55 3628800       1      10
> range(x)
[1]  1 10
```

These four functions can also be used on more than one vector, in which case the sum, product, minimum, or maximum are taken over all elements of all vectors.

```
> y <- 11:20
> sum(x, y)
[1] 210
> prod(x, y)
[1] 2.4329e+18
> max(x, y)
[1] 20
> min(x, y)
[1] 1
```

1

What does the function `any()` do? The function answers the question: Given a set of logical vectors, is at least one of the values true?
The function `all()` is the complement of `any()`; it answers the question: Given a set of logical vectors, are all of the values true?

Cumulative vector operation

The function `cumsum()` belongs to a group of four functions which computes cumulative sums, products, `cumprod()`, and extremes, `cummin()`, `cummax()`.
Cumulating a Vector: 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.

```
> set.seed(1848)
> x <- round(rnorm(10), 2)
> x
 [1] -1.20  1.84  0.92 -0.70  1.93 -1.32 -0.97  0.18 -2.25  0.78
> cumsum(x)
 [1] -1.20  0.64  1.56  0.86  2.79  1.47  0.50  0.68 -1.57 -0.79
> round(cumprod(x), 2)
 [1] -1.20 -2.21 -2.03  1.42  2.74 -3.62  3.51  0.63 -1.42 -1.11
```

[1] The semicolon was used here to separate more than one command typed in on the same line.

```
> cummin(x)
 [1] -1.20 -1.20 -1.20 -1.20 -1.20 -1.32 -1.32 -1.32 -2.25 -2.25
> cummax(x)
 [1] -1.20  1.84  1.84  1.84  1.93  1.93  1.93  1.93  1.93  1.93
```

Sorting and ordering vectors

Sorting a vector: To sort a vector in increasing order, use the function `sort()` [2]. You can also use this function to sort in decreasing order by using the argument `decrease = TRUE`.

```
> x1 <- c(2, 6, 4, 5, 5, 8, 8, 1, 3, 0)
> length(x1)
[1] 10
> x2 <- sort(x1)
> x3 <- sort(x1, decreasing = TRUE)
```

Ordering a vector: 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)
> y <- 1:10
> z <- order(x)
```

Sort

```
> sort(x)
 [1] -2.03190 -1.66859 -1.54659 -0.81197 -0.66099 -0.50070 -0.30437 -0.17851
 [9]  0.27800  0.35339
```

Change the order of elements of y

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

Try to figure out what the result of `x[order(x)]` is!
Reversing a vector: The function `rev` reverses the order of vector elements. `rev(sort(x))` is a sorted vector in descending order.

```
> x <- round(rnorm(10), 2)
> rev(sort(x))
 [1]  1.12  1.02  1.01  0.66  0.59  0.52 -0.42 -0.53 -0.73 -2.02
```

[2] Note that `sort()` returns a new vector that contains the sorted elements of the original vector; it does not sort the original vector.

Making vectors unique

The function `unique` returns a vector which only contains the unique values of the input vector. The function `duplicated()` returns `TRUE` or `FALSE` for every element 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
```

Differencing a vector

Our last example of a vector manipulation function is the function `diff`. This 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
```

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.

2.2 MANIPULATING MATRICES

Subsetting a matrix

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 can take different forms.

Subsetting by rows and columns integers

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

```
> X <- matrix(1:36, ncol = 6)
> X[2, 6]
```

```
[1] 32
```

The third row:

```
> 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` without 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 original matrix `X` remains the same, unless you assign the result back to `X`.

```
> X <- X[-3, 4]
> X
```

```
[1] 19 20 22 23 24
```

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 only the element of the first row, second column:

```
> X <- matrix(1:36, ncol = 6)
> X[1, 2] <- 5
```

To change a complete column:

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

Subsetting by a logical matrix

We can also subset a matrix by 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, to replace all elements greater than 0 with 0:

```
> X <- matrix(rnorm(100), ncol = 10)
> X[X > 0] <- 0
> X
          [,1]     [,2]      [,3]     [,4]      [,5]      [,6]     [,7]
 [1,] -0.27419  0.00000 -1.513568 -1.46420 -0.683043 -0.181865  0.00000
 [2,] -0.77971  0.00000 -1.007833 -0.14746 -1.128581 -0.023738 -1.63358
 [3,]  0.00000  0.00000 -0.582485  0.00000 -0.906651 -0.276996 -0.11921
 [4,]  0.00000 -0.20357  0.000000 -0.29551 -0.084761 -0.273663  0.00000
 [5,] -0.91790  0.00000  0.000000 -0.56061  0.000000  0.000000 -0.76240
 [6,]  0.00000 -0.34021 -0.142731 -0.69350 -0.264546 -0.728286  0.00000
 [7,]  0.00000  0.00000  0.000000 -1.55097 -0.167526  0.000000  0.00000
 [8,]  0.00000 -1.58287  0.000000 -2.09688  0.000000  0.000000  0.00000
 [9,] -0.30224 -1.12021 -0.053255 -0.30930  0.000000  0.000000 -0.57624
[10,]  0.00000 -2.70392 -1.041294 -0.77341 -0.389621  0.000000  0.00000
          [,8]      [,9]    [,10]
 [1,]  0.00000  0.000000  0.00000
 [2,] -0.11297 -0.111318 -0.31460
 [3,]  0.00000 -0.010181 -0.56983
```

```
 [4,] -0.92090 -1.680854  0.00000
 [5,]  0.00000 -1.453314 -0.54376
 [6,]  0.00000 -1.599849  0.00000
 [7,] -1.90633 -0.401601  0.00000
 [8,]  0.00000 -2.177284 -0.23241
 [9,] -0.96684  0.000000 -1.36440
[10,]  0.00000 -0.141607 -0.40021
```

Subsetting with two columns

We can also subset a matrix X with two columns. A row of X consists of two numbers, each row of X 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
> INDEX <- cbind(c(1, 2, 5), c(3, 4, 4))
> INDEX
     [,1] [,2]
[1,]    1    3
[2,]    2    4
[3,]    5    4
> X[INDEX]
[1] 13 20 23
```

Subsetting by a single number or a vector of numbers

What happens when we subset a matrix by a single number or one vector of numbers? In this case the matrix is treated as a vector where all the columns are stacked.

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

2.3 MANIPULATING DATA FRAMES

Although a data frame can be considered as a vector of lists, it shares common subsetting methods as matrices. But data frames also offer a few extra possibilities which will be considered in the following.

Extracting data from data frames

As an example let us consider Longley's US Economic data set

```
> longley
     GNP.deflator    GNP Unemployed Armed.Forces Population Year Employed
1947         83.0 234.29      235.6        159.0     107.61 1947   60.323
1948         88.5 259.43      232.5        145.6     108.63 1948   61.122
1949         88.2 258.05      368.2        161.6     109.77 1949   60.171
1950         89.5 284.60      335.1        165.0     110.93 1950   61.187
1951         96.2 328.98      209.9        309.9     112.08 1951   63.221
1952         98.1 347.00      193.2        359.4     113.27 1952   63.639
1953         99.0 365.38      187.0        354.7     115.09 1953   64.989
1954        100.0 363.11      357.8        335.0     116.22 1954   63.761
1955        101.2 397.47      290.4        304.8     117.39 1955   66.019
1956        104.6 419.18      282.2        285.7     118.73 1956   67.857
1957        108.4 442.77      293.6        279.8     120.44 1957   68.169
1958        110.8 444.55      468.1        263.7     121.95 1958   66.513
1959        112.6 482.70      381.3        255.2     123.37 1959   68.655
1960        114.2 502.60      393.1        251.4     125.37 1960   69.564
1961        115.7 518.17      480.6        257.2     127.85 1961   69.331
1962        116.9 554.89      400.7        282.7     130.08 1962   70.551
```

To extract the column names of the data frame use the function `names()`

```
> names(longley)
[1] "GNP.deflator" "GNP"          "Unemployed"   "Armed.Forces" "Population"
[6] "Year"         "Employed"
```

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

```
> GNP <- longley$GNP
> GNP
 [1] 234.29 259.43 258.05 284.60 328.98 347.00 365.38 363.11 397.47 419.18
[11] 442.77 444.55 482.70 502.60 518.17 554.89
> GNP <- longley[["GNP"]]
> GNP
 [1] 234.29 259.43 258.05 284.60 328.98 347.00 365.38 363.11 397.47 419.18
[11] 442.77 444.55 482.70 502.60 518.17 554.89
> class(GNP)
[1] "numeric"
```

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

```
> GNP <- longley["GNP"]
> GNP
        GNP
1947 234.29
1948 259.43
1949 258.05
1950 284.60
1951 328.98
1952 347.00
1953 365.38
1954 363.11
1955 397.47
1956 419.18
1957 442.77
1958 444.55
1959 482.70
1960 502.60
1961 518.17
1962 554.89
> class(GNP)
[1] "data.frame"
```

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

```
> longley[, c("GNP", "Population")]
        GNP Population
1947 234.29     107.61
1948 259.43     108.63
1949 258.05     109.77
1950 284.60     110.93
1951 328.98     112.08
1952 347.00     113.27
1953 365.38     115.09
1954 363.11     116.22
1955 397.47     117.39
1956 419.18     118.73
1957 442.77     120.44
1958 444.55     121.95
1959 482.70     123.37
1960 502.60     125.37
1961 518.17     127.85
1962 554.89     130.08
```

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

```
> Year1960 <- longley["1960", ]
> Year1960
     GNP.deflator   GNP Unemployed Armed.Forces Population Year Employed
1960        114.2 502.6      393.1        251.4     125.37 1960   69.564
> class(Year1960)
[1] "data.frame"
```

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

```
> longley[c("1955", "1960"), ]
     GNP.deflator    GNP Unemployed Armed.Forces Population Year Employed
1955        101.2 397.47      290.4        304.8     117.39 1955   66.019
1960        114.2 502.60      393.1        251.4     125.37 1960   69.564
```

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

```
> longley[c("1955", "1960", "1965"), ]
     GNP.deflator    GNP Unemployed Armed.Forces Population Year Employed
1955        101.2 397.47      290.4        304.8     117.39 1955   66.019
1960        114.2 502.60      393.1        251.4     125.37 1960   69.564
NA             NA     NA         NA           NA         NA   NA       NA
```

Rows from a data frame can also be selected using row numbers.

```
> longley[5:10, ]
     GNP.deflator    GNP Unemployed Armed.Forces Population Year Employed
1951         96.2 328.98      209.9        309.9     112.08 1951   63.221
1952         98.1 347.00      193.2        359.4     113.27 1952   63.639
1953         99.0 365.38      187.0        354.7     115.09 1953   64.989
1954        100.0 363.11      357.8        335.0     116.22 1954   63.761
1955        101.2 397.47      290.4        304.8     117.39 1955   66.019
1956        104.6 419.18      282.2        285.7     118.73 1956   67.857
```

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

```
> head(longley, 3)
     GNP.deflator    GNP Unemployed Armed.Forces Population Year Employed
1947         83.0 234.29      235.6        159.0     107.61 1947   60.323
1948         88.5 259.43      232.5        145.6     108.63 1948   61.122
1949         88.2 258.05      368.2        161.6     109.77 1949   60.171


> tail(longley, 2)
     GNP.deflator    GNP Unemployed Armed.Forces Population Year Employed
1961        115.7 518.17      480.6        257.2     127.85 1961   69.331
1962        116.9 554.89      400.7        282.7     130.08 1962   70.551
```

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 to a TRUE are selected. Suppose you want to get all stock exchanges from the Longley data frame that have a GNP of over 350. First create a logical vector `index`:

```
> index <- longley$GNP > 350
> index
 [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[13]  TRUE  TRUE  TRUE  TRUE
```

Now use this vector to subset the data frame:

```
> longley[index, ]
     GNP.deflator    GNP Unemployed Armed.Forces Population Year Employed
1953         99.0 365.38      187.0        354.7     115.09 1953   64.989
1954        100.0 363.11      357.8        335.0     116.22 1954   63.761
1955        101.2 397.47      290.4        304.8     117.39 1955   66.019
1956        104.6 419.18      282.2        285.7     118.73 1956   67.857
1957        108.4 442.77      293.6        279.8     120.44 1957   68.169
1958        110.8 444.55      468.1        263.7     121.95 1958   66.513
1959        112.6 482.70      381.3        255.2     123.37 1959   68.655
1960        114.2 502.60      393.1        251.4     125.37 1960   69.564
1961        115.7 518.17      480.6        257.2     127.85 1961   69.331
1962        116.9 554.89      400.7        282.7     130.08 1962   70.551
```

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

```
> subset(longley, GNP > 350 & Population > 110)
     GNP.deflator    GNP Unemployed Armed.Forces Population Year Employed
1953         99.0 365.38      187.0        354.7     115.09 1953   64.989
1954        100.0 363.11      357.8        335.0     116.22 1954   63.761
1955        101.2 397.47      290.4        304.8     117.39 1955   66.019
1956        104.6 419.18      282.2        285.7     118.73 1956   67.857
1957        108.4 442.77      293.6        279.8     120.44 1957   68.169
1958        110.8 444.55      468.1        263.7     121.95 1958   66.513
1959        112.6 482.70      381.3        255.2     123.37 1959   68.655
1960        114.2 502.60      393.1        251.4     125.37 1960   69.564
1961        115.7 518.17      480.6        257.2     127.85 1961   69.331
1962        116.9 554.89      400.7        282.7     130.08 1962   70.551
```

Adding columns to a data frame

The function `cbind` can be used to add additional columns to a data frame. For example, the ratio of the GNP and the population

```
> gnpPop <- round(longley[, "GNP"]/longley[, "Population"], 2)
> longley <- cbind(longley, GNP.POP = gnpPop)
> longley
     GNP.deflator    GNP Unemployed Armed.Forces Population Year Employed
1947         83.0 234.29      235.6        159.0     107.61 1947   60.323
1948         88.5 259.43      232.5        145.6     108.63 1948   61.122
1949         88.2 258.05      368.2        161.6     109.77 1949   60.171
1950         89.5 284.60      335.1        165.0     110.93 1950   61.187
1951         96.2 328.98      209.9        309.9     112.08 1951   63.221
1952         98.1 347.00      193.2        359.4     113.27 1952   63.639
1953         99.0 365.38      187.0        354.7     115.09 1953   64.989
1954        100.0 363.11      357.8        335.0     116.22 1954   63.761
1955        101.2 397.47      290.4        304.8     117.39 1955   66.019
```

```
1956        104.6 419.18      282.2        285.7     118.73 1956   67.857
1957        108.4 442.77      293.6        279.8     120.44 1957   68.169
1958        110.8 444.55      468.1        263.7     121.95 1958   66.513
1959        112.6 482.70      381.3        255.2     123.37 1959   68.655
1960        114.2 502.60      393.1        251.4     125.37 1960   69.564
1961        115.7 518.17      480.6        257.2     127.85 1961   69.331
1962        116.9 554.89      400.7        282.7     130.08 1962   70.551
     GNP.POP
1947    2.18
1948    2.39
1949    2.35
1950    2.57
1951    2.94
1952    3.06
1953    3.17
1954    3.12
1955    3.39
1956    3.53
1957    3.68
1958    3.65
1959    3.91
1960    4.01
1961    4.05
1962    4.27
```

```
> class(longley)
```

```
[1] "data.frame"
```

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

Combining data frames

Use the function `rbind` to combine (or stack) two or more data frames.

Merging data frames

Two data frames can be merged into one data frame using the function `merge`. [3] If the original data frames contain identical columns, these columns only appear once in the merged data frame. Consider the following two data frames:

```
> long1 <- longley[1:6, c("Year", "Population", "Armed.Forces")]
> long1
```

```
     Year Population Armed.Forces
1947 1947     107.61        159.0
1948 1948     108.63        145.6
1949 1949     109.77        161.6
1950 1950     110.93        165.0
1951 1951     112.08        309.9
1952 1952     113.27        359.4
```

[3] In database terminology, this is the join operation.

```
> long2 <- longley[1:6, c("Year", "GNP", "Unemployed")]
> long2
     Year    GNP Unemployed
1947 1947 234.29      235.6
1948 1948 259.43      232.5
1949 1949 258.05      368.2
1950 1950 284.60      335.1
1951 1951 328.98      209.9
1952 1952 347.00      193.2
> long3 <- merge(long1, long2)
> long3
  Year Population Armed.Forces    GNP Unemployed
1 1947     107.61        159.0 234.29      235.6
2 1948     108.63        145.6 259.43      232.5
3 1949     109.77        161.6 258.05      368.2
4 1950     110.93        165.0 284.60      335.1
5 1951     112.08        309.9 328.98      209.9
6 1952     113.27        359.4 347.00      193.2
```

By default the `merge` function leaves out rows that were 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 right 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 0.70654       45
2    7 0.99467       89
3    9 0.11259       14
4  110      NA       90
```

For more complex examples see the help file of the function `merge()`.

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 has the type factor, and numerical calculations cannot 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
```

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 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("M3", "GDP"))
```

2.4 WORKING WITH 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.

```
> set.seed(4711)
> x <- rnorm(12)
> 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, "seed") <- "seed = 4711"
> x
 [1]  1.819735  1.370440  1.196318 -0.406879 -0.610979 -1.508912  0.817549
 [8] -0.964668 -0.044522  0.474355 -0.982166 -1.572111
attr(,"seed")
[1] "seed = 4711"
> attr(x, "seed")
```

```
[1] "seed = 4711"
```

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 = length(x), atr2 = "length")
> x
 [1]  1.819735  1.370440  1.196318 -0.406879 -0.610979 -1.508912  0.817549
 [8] -0.964668 -0.044522  0.474355 -0.982166 -1.572111
attr(,"seed")
[1] "seed = 4711"
attr(,"atr1")
[1] 12
attr(,"atr2")
[1] "length"
```

```
> attr(x, "seed")
```

```
[1] "seed = 4711"
```

```
> attr(x, "atr1")
```

```
[1] 12
```

```
> attr(x, "atr2")
```

```
[1] "length"
```

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)
```

```
$seed
[1] "seed = 4711"

$atr1
[1] 12

$atr2
[1] "length"
```

In order to get the description attribute of x use:

```
> attributes(x)$seed
```

```
[1] "seed = 4711"
```

Or type in the following construction:

```
> attr(x, "seed")
```

```
[1] "seed = 4711"
```

We mentioned earlier that matrices are just vectors with a dimension attribute. We can know inspect this attribute and understand why we can also create matrices from vectors by using the `dim()` function.

```
> m <- matrix(1:4, ncol = 2)
> attributes(m)
$dim
[1] 2 2

> v <- 1:4
> dim(v) <- c(2, 2)
> attributes(v)

$dim
[1] 2 2
```

2.5 MANIPULATING CHARACTER STRINGS

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

The functions `nchar()`, `substring()` and `paste()`

```
> charvector <- c("1970 | 1,003.2 | 4.11 | 6.21 Mio", "1975 | 21,034.6  | 5.31 | 7.11 Mio",
     "1980 | 513.2 | 4.79 |7.13 Mio")
> charvector
[1] "1970 | 1,003.2 | 4.11 | 6.21 Mio"   "1975 | 21,034.6  | 5.31 | 7.11 Mio"
[3] "1980 | 513.2 | 4.79 |7.13 Mio"
```

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

```
> nchar(charvector)
[1] 32 34 29
```

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

```
> years <- substring(charvector, first = 1, last = 4)
> years
[1] "1970" "1975" "1980"
```

The function `paste` will paste two or more character objects. For example, to create a character vector with: "Year-12-31"

```
> paste(years, "12", "31", sep = "-")
[1] "1970-12-31" "1975-12-31" "1980-12-31"
```

The argument `sep` is used to specify the separating symbol between the two character objects. Use `sep = ""` for no space between the character objects.

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 patterns. See the help page for `regexpr()` to find out more about regular expressions.
Let's extract the row names from our Longley data frame.

```
> longleyNames <- names(longley)
```

We want to know if a string in `longleyNames` contains the pattern `"GNP"` and if this is true we want to know the column numbers. To do this we can use the function `grep()`:

```
> index <- grep("GNP", longleyNames)
> index
[1] 1 2 8
```

As we can see from the output, elements 1, 2, 8 of the `longleyNames` vector are names containing the string 'GNP', which is confirmed by a quick check:

```
> longleyNames[index]
[1] "GNP.deflator" "GNP"          "GNP.POP"
```

We can also extract all names starting with the letter 'G', using a regular expression:

```
> index <- grep("^G", longleyNames)
> longleyNames[index]
[1] "GNP.deflator" "GNP"          "GNP.POP"
```

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

```
> gnpMatch <- regexpr("GNP", longleyNames)
> gnpMatch
[1]  1  1 -1 -1 -1 -1 -1  1
attr(,"match.length")
[1]  3  3 -1 -1 -1 -1 -1  3
```

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, with the `match.length` attribute indicating the length of the matching string. In our example we see that all but the elements 3,4,5,6 and 7 are equal to one, which means that `GNP` is part of these column names. Again, a quick check:

```
> longleyNames[index]
```

```
[1] "GNP.deflator" "GNP"          "GNP.POP"
```

If character vectors become too long to see the match quickly, use the following trick:

```
> index <- 1:length(longleyNames)
> index[gnpMatch > 0]
[1] 1 2 8
```

The result of the function `regexpr()` contains the attribute `match.length`, which gives the length of the matched text. In the above example all mathed strings consist of 3 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
> attr(w, "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".
Now we can use the `substring()` function to extract the integers. `substring()` takes a character vector, start and stop values as arguments. Our start values are given by `w`, and to get the stop values, we simply need to add the `match.length` to the start values, and subtract 1. Note that the result of the substring function has the 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
```

Replacing characters

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

```
> gsub(".POP", "/Population", longleyNames)
```

```
[1] "GNP.deflator"   "GNP"            "Unemployed"     "Armed.Forces"
[5] "Population"     "Year"           "Employed"       "GNP/Population"
```

Note that by default, the `pattern` argument is a regular expression. If 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."
```

Splitting character strings

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`.

```
> charvector <- c("1970 | 1,003.2 | 4.11 | 6.21 Mio", "1975 | 21,034.6  | 5.31 | 7.11 Mio",
     "1980 | 513.2 | 4.79 |7.13 Mio")
> mysplit <- strsplit(x = charvector, split = "\\|")
> mysplit
```

```
[[1]]
[1] "1970 "     " 1,003.2 " " 4.11 "    " 6.21 Mio"

[[2]]
[1] "1975 "       " 21,034.6  " " 5.31 "      " 7.11 Mio"

[[3]]
[1] "1980 "    " 513.2 "  " 4.79 "   "7.13 Mio"
```

Now extract the third column und convert to numerical values

```
> unlisted <- unlist(mysplit)
> unlisted
```

```
 [1] "1970 "       " 1,003.2 "   " 4.11 "      " 6.21 Mio"   "1975 "
 [6] " 21,034.6  " " 5.31 "      " 7.11 Mio"   "1980 "       " 513.2 "
[11] " 4.79 "      "7.13 Mio"

> as.numeric(unlisted[seq(3, length(unlisted), by = 4)])

[1] 4.11 5.31 4.79
```

2.6 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. As an example, let us break the average daily turnover of the stock markets into logarithmic equidistant units

```
> GNP <- longley[, "GNP"]
> breaks <- (2:6) * 100
> cut(x = GNP, breaks)
 [1] (200,300] (200,300] (200,300] (200,300] (300,400] (300,400] (300,400]
 [8] (300,400] (300,400] (400,500] (400,500] (400,500] (400,500] (500,600]
[15] (500,600] (500,600]
Levels: (200,300] (300,400] (400,500] (500,600]
```

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

```
> cut(x = GNP, breaks = 3)
 [1] (234,341] (234,341] (234,341] (234,341] (234,341] (341,448] (341,448]
 [8] (341,448] (341,448] (341,448] (341,448] (341,448] (448,555] (448,555]
[15] (448,555] (448,555]
Levels: (234,341] (341,448] (448,555]
```

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

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

> class(Levels)

[1] "factor"


> data.frame(GNP = longley[, "GNP"], Level = as.vector(Levels))
      GNP  Level
1  234.29    low
2  259.43    low
3  258.05    low
4  284.60    low
5  328.98    low
6  347.00 medium
7  365.38 medium
8  363.11 medium
```

```
9  397.47 medium
10 419.18 medium
11 442.77 medium
12 444.55 medium
13 482.70   high
14 502.60   high
15 518.17   high
16 554.89   high
```

CHAPTER 3

IMPORTING AND EXPORTING DATA

One of the first things you want to do in a statistical data analysis system is to import data and to save the results. R provides a few methods to import and export data. These are the subject of this chapter.

3.1 WRITING TO TEXT FILES

We will start by writing some data to a text file, and then, later, we will import the data from this file.
Suppose we have the following text

```
(C) Alabini AG
Date: 18-05-2009
Comments: Class III Products
ProductID,   Price, Quality,       Company
    23851, 1245.30,       A,   Mercury Ltd
     3412,  941.40,      BB,      Flury SA
    12184, 1499.00,      AA, Inkoa Holding
```

which we want to write to a file using the R function `write()`.

```
> args(write)
function (x, file = "data", ncolumns = if (is.character(x)) 1 else 5,
    append = FALSE, sep = " ")
NULL
```

The first argument `x` is the data to be written to the file, the second argument `file` is the path, or a character string naming the file to write to, `ncolumns` the number of columns to write the data in. The `append` argument, if set to `TRUE`, specifies whether the data `x` are appended to the connection or not, and the `sep` argument specifies the string used to separate the columns.

```
> headerLines <- c("(C) Alabini AG", "Date: 18-05-2009", "Comment: Class III Products")
```

```
> recordLines <- c("ProductID,   Price, Quality, Company", "    23851, 1245.30,       A, Mercury Ltd",
     "     3412,  941.40,      BB, Flury SA", "    12184, 1499.00,      AA, Inkoa Holding")


> file <- "alabini.txt"
> write(headerLines, file)
> write(recordLines, file, append = TRUE)
```

if the `file` name is empty, file = "", the `write()` prints to the standard output connection, i.e. the console:

```
> write(c(headerLines, recordLines), file = "")
(C) Alabini AG
Date: 18-05-2009
Comment: Class III Products
ProductID,   Price, Quality, Company
    23851, 1245.30,       A, Mercury Ltd
     3412,  941.40,      BB, Flury SA
    12184, 1499.00,      AA, Inkoa Holding
```

3.2 READING FROM A TEXT FILE WITH `scan()`

The function `scan()` reads a text file element by element. This can be, for example, word by word or line by line.

```
> args(scan)
function (file = "", what = double(0), nmax = -1, n = -1, sep = "",
    quote = if (identical(sep, "\n")) "" else "'\"", dec = ".",
    skip = 0, nlines = 0, na.strings = "NA", flush = FALSE, fill = FALSE,
    strip.white = FALSE, quiet = FALSE, blank.lines.skip = TRUE,
    multi.line = TRUE, comment.char = "", allowEscapes = FALSE,
    encoding = "unknown")
NULL
```

LISTING 3.1: SELECTED ARGUMENTS FOR THE FUNCTION SCAN

```
Argument:
file        name of a file to read data values from
what        type of what gives the type of data to be read
nmax        maximum number of data values to be read
n           number of data values to be read
sep         what to read as delimited input fields
quote       quoting characters as a single character string or NULL
dec         decimal point character
skip        number of lines to skip before beginning to read
nlines      if positive, maximum number of lines to be read
```

By default the function expects objects of type `double` to be read in. In our example we have to change the argument `what` to `character(0)`.

```
> scan("alabini.txt", what = character(0))
 [1] "(C)"        "Alabini"    "AG"         "Date:"      "18-05-2009"
 [6] "Comment:"   "Class"      "III"        "Products"   "ProductID,"
[11] "Price,"     "Quality,"   "Company"    "23851,"     "1245.30,"
[16] "A,"         "Mercury"    "Ltd"        "3412,"      "941.40,"
[21] "BB,"        "Flury"      "SA"         "12184,"     "1499.00,"
[26] "AA,"        "Inkoa"      "Holding"
```

What we get is not what we wanted. We have still to specify the field separator to be set to newlines

```
> scan("alabini.txt", what = character(0), sep = "\n")
[1] "(C) Alabini AG"
[2] "Date: 18-05-2009"
[3] "Comment: Class III Products"
[4] "ProductID,   Price, Quality, Company"
[5] "    23851, 1245.30,       A, Mercury Ltd"
[6] "     3412,  941.40,      BB, Flury SA"
[7] "    12184, 1499.00,      AA, Inkoa Holding"
```

3.3 READING FROM A TEXT FILE WITH readLines()

To read lines from a text file, we can also use the function `readLines()`. The function reads some or all text lines from a connection and assumes text lines and a newline character at the end of each line by default.

```
> args(readLines)
function (con = stdin(), n = -1L, ok = TRUE, warn = TRUE, encoding = "unknown")
NULL
```

LISTING 3.2: SELECTED ARGUMENTS FOR THE FUNCTION READLINES()

```
Argument:
con         a connection object or a character string
n           an integer, the (maximal) number of lines to read
encoding    encoding to be assumed for input strings
```

Calling `readLines()` on our example returns:

```
> readLines("alabini.txt")
[1] "(C) Alabini AG"
[2] "Date: 18-05-2009"
[3] "Comment: Class III Products"
[4] "ProductID,   Price, Quality, Company"
[5] "    23851, 1245.30,       A, Mercury Ltd"
[6] "     3412,  941.40,      BB, Flury SA"
[7] "    12184, 1499.00,      AA, Inkoa Holding"
```

3.4 READING FROM A TEXT FILE WITH `read.table()`

If we want to import just the data part as a data frame, we can use the function `read.table()` and skip the header lines. The function has a whole bundle of arguments, e.g. to specify the header, the column separator, the number of lines to skip, the data types of the columns, etc.

```
> args(read.table)
function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
    row.names, col.names, as.is = !stringsAsFactors, na.strings = "NA",
    colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
    fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
    comment.char = "#", allowEscapes = FALSE, flush = FALSE,
    stringsAsFactors = default.stringsAsFactors(), fileEncoding = "",
    encoding = "unknown")
NULL
```

LISTING 3.3: SELECTED ARGUMENTS FOR THE FUNCTION `read.table()`

```
Argument:
file             the name of the file can also be a URL
sep              the field separator character
quote            the set of quoting characters
dec              the character used for decimal points
row.names        a vector of row names
col.names        a vector of optional names for the variables
colClasses       vector of classes to be assumed for the columns
nrows            maximum number of rows to read in
skip             number of lines to skip before beginning to read
stringsAsFactors should character vectors be converted to factors?
```

The function reads a file in table format and creates a data frame from it, with cases corresponding to rows and variables to columns in the file. Now let us read in our example data file. Remember to skip the first three lines, set the header to true and set the field separator to a comma.

```
> alabini <- read.table("alabini.txt", skip = 3, header = TRUE,
     sep = ",")
> alabini
  ProductID  Price  Quality        Company
1     23851 1245.3        A    Mercury Ltd
2      3412  941.4       BB       Flury SA
3     12184 1499.0       AA  Inkoa Holding

> class(alabini)

[1] "data.frame"
```

The returned object from the function `read.table()` is a data.frame, but what are the classes of the columns?

```
> Classes <- c(class(alabini[, 1]), class(alabini[, 2]), class(alabini[,
    3]), class(alabini[, 4]))
> names(Classes) = names(alabini)
> Classes
```

```
ProductID     Price   Quality   Company
"integer" "numeric"  "factor"  "factor"
```

The first solumn is an object of class `integer`, the second of class `numeric` and the last two columns are `factors`.

Character versus factor input columns

By default, R converts character data in text files into the type factor. In the above example, the third and fourth columns are factors. If you want to keep character data as character data in R, use the `stringsAsFactors` argument, and set it to FALSE.

```
> alabini <- read.table("alabini.txt", skip = 3, header = TRUE,
    sep = ",", stringsAsFactors = FALSE)
```

```
> Classes <- c(class(alabini[, 1]), class(alabini[, 2]), class(alabini[,
    3]), class(alabini[, 4]))
> names(Classes) = names(alabini)
> Classes
```

```
  ProductID       Price     Quality     Company
  "integer"   "numeric" "character" "character"
```

Specifying the input classes of columns

To specify that certain columns are characters and other columns are not you can use the `colClasses` argument and provide the type for each column. As an example, we want use the quality as a factor variable and the company names as characters.

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

```
> Classes <- c(class(alabini[, 1]), class(alabini[, 2]), class(alabini[,
    3]), class(alabini[, 4]))
> names(Classes) = names(alabini)
> Classes
```

```
  ProductID       Price     Quality     Company
  "numeric"   "numeric"    "factor" "character"
```

Reading quoted strings from input

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. To import such text files use the `quote` argument. Suppose we have the following comma separated text file that we want to read.

```
> headerLines <- c("(C) Alabini AG", "Date: 18-05-2009", "Comment: Class III Products")


> recordLines <- c(
     "ProductID,   Price, Quality, Company",
     "    23851, 1245.30,       A, 'Mercury Ltd'",
     "     3412,  941.40,      BB, 'Flury SA'",
     "    12184, 1499.00,      AA, 'Inkoa Holding'")


> file <- "alabiniQuoted.txt"
> write(headerLines, file)
> write(recordLines, file, append = TRUE)
```

Have a look at the difference.

```
> read.table("alabiniQuoted.txt", skip = 3, sep = ",", header = TRUE,
     stringsAsFactors = FALSE)
  ProductID  Price  Quality        Company
1     23851 1245.3        A    Mercury Ltd
2      3412  941.4       BB       Flury SA
3     12184 1499.0       AA  Inkoa Holding


> read.table("alabiniQuoted.txt", skip = 3, sep = ",", header = TRUE,
     stringsAsFactors = FALSE, quote = "")
  ProductID  Price  Quality          Company
1     23851 1245.3        A    'Mercury Ltd'
2      3412  941.4       BB       'Flury SA'
3     12184 1499.0       AA  'Inkoa Holding'
```

Reading CSV files

If you look in the help file of the `read.table()` function you will find four more functions with tailored arguments for special file types such as CSV files, which are really just wrappers for `read.table()`.
The functions `read.csv()`, `read.csv2()`, `read.delim()`, and `read.delim2()` have specific arguments. For example, in the function `read.csv()`, the default value of the `sep` argument is a comma for commata separated files, in the function `read.csv2()` the default value is a semicolon, taking into account country-specific CSV file separators, and the functions `read.delim()` and `read.delim2()` have a tab character, `"\t"`, as the default. For further specific settings we refer to the help page.

3.5 IMPORTING EXAMPLE DATA FILES

The function `data()` loads specified data sets, or lists the available data sets.
Four formats of data files are supported:

LISTING 3.4: EXAMPLE DATA FILE FORMATS

```
Argument:
.R, .r                  these files are read with source() with the
                        working  directory changed temporarily to
                        the directory containing the respective file
.RData, .rd             these files are read with the function load()
.tab, .txt, .TXT        these files are read with the function
                        read.table(..., header = TRUE)}, and hence
                        result in a data frame.
.csv, .CSV              these files are read with the function
                        read.table(..., header = TRUE, sep = ";")
                        also result in a data frame.
```

If more than one matching file name is found, the first on this list is used.

Example: Get Euro conversion rate

R has an example data set with the Euro foreign exchange conversion rates.

```
> data(euro)
> euro
       ATS        BEF        DEM        ESP        FIM        FRF        IEP
  13.76030   40.33990    1.95583  166.38600    5.94573    6.55957    0.78756
       ITL        LUF        NLG        PTE
1936.27000   40.33990    2.20371  200.48200

> class(euro)

[1] "numeric"
```

Example: UK gas consumption

Another example data set hold quarterly UK gas consumption data in millions of therms.

```
> data(UKgas)
> head(UKgas)
[1] 160.1 129.7  84.8 120.1 160.1 124.9

> class(UKgas)

[1] "ts"
```

3.6 IMPORTING HISTORICAL DATA SETS FROM THE INTERNET

As we have already seen, the first argument in the functions `readLines()` and `read.table()` must not necessarily a file name it can als be a connection to the Internet. On the Internet we can find several financial data source which we can use from R for our statistical analysis. Two of the sources we will present here: (i) the FRED2 database from the Federal Reserve in St. Louis which has a huge database from daily to annual data sets of economic time series, and (ii) Yahoo Finance with thousands of financial time series, including stock market indices, equitiy prices, or exchange traded funds amongst others

CHAPTER 4

OBJECT TYPES

In this chapter we provide a preliminary description of the various data types which are provided by R. More detailed discussions of many of them will be found in the subsequent chapters.

4.1 CHARACTERIZATION OF OBJECTS

Objects are characterized by their *type*, by their *storage mode* and by their object *class*.

The function `typeof()`

To identify the type of an R object you can use the R function `typeof()`, which returns the type of an R object.
The following table lists the most prominent values as returned by the function `typeof()`

LISTING 4.1: THE MOST COMMON OBJECT TYPES

```
Object Type:
double          a vector containing real values
integer         a vector containing integer values
complex         a vector containing complex values
logical         a vector containing logical values
character       a vector containing character values
NULL            NULL
```

LISTING 4.2: LESS COMMON OBJECT TYPES

```
Object Type:
any             a special type that matches all types
builtin         an internal function that evaluates its arguments
```

```
closure         a function
environment     an environment
expression      an expression object
externalptr     an external pointer object
language        an R language construct
pairlist        a pairlist object
promise         an object used to implement lazy evaluation
raw             a vector containing bytes
S4              an S4 object which is not a simple object
special         internal function that does not evaluate its arguments
symbol          a variable name
weakref         a weak reference object
...             the special variable length argument
```

The storage mode of a function

Both functions `mode()` and `storage.mode()` get or set the storage mode of an object.
Modes have the same set of names as types except that

- types "integer" and "double" are returned as "numeric"
- types "special" and "builtin" are returned as "function"
- type "symbol" is called mode "name"
- type "language" is returned as "`(`" or "call"

The mode is generally used when calling functions written in another language, such as C or FORTRAN, to ensure that `R` objects have the data type expected by the routine being called. Note that in the S language, vectors with integer or real values are both of mode "numeric", so their storage modes need to be distinguished.

The class of an object

`R` possesses a simple generic function mechanism which can be used for an object-oriented style of programming. Method dispatch on appropriate function to a generic function based on the class of the first argument for S3 methods, and on any argument for S4 methods. The `R` function `class()` returns the name of the object class to which the object belongs.

4.2 DOUBLE

The type of double appears manifold in different `R` objects. These include real numbers, infinite values, and date and time objects.

Real numbers

If you perform calculations on (real) numbers, you can use the data type double to represent the numbers. Doubles are numbers, such as `314.15`, `1.0014` and `1.0019`. Doubles are used to represent continuous variables such as prices or financial returns.

```
> bid <- 1.0014
> ask <- 1.0019
> spread <- ask - bid
```

Use the function `is.double()` to check whether an object is of type double.

```
> is.double(spread)
[1] TRUE
```

Alternatively, use the function `typeof()` to obtain the type of the object `spread`.

```
> object <- spread
> c(typeof = typeof(object), mode = mode(object), storag.mode = storage.mode(object),
     class = class(object))
     typeof        mode storag.mode       class
   "double"   "numeric"    "double"   "numeric"
```

Infinite Values

Infinite values are represented by `Inf` or `-Inf`. The type of these values is double.

```
> object <- Inf
> c(typeof = typeof(object), mode = mode(object), storag.mode = storage.mode(object),
     class = class(object))
     typeof        mode storag.mode       class
   "double"   "numeric"    "double"   "numeric"
```

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
```

Date objects

To represent a calendar date in R use the function `as.Date` to create an object of class `Date`. Calendar dates can be generated from character scalars or vectors, for example.

```
> timeStamps <- c("1973-12-09", "1974-08-29")
> Date <- as.Date(timeStamps, "%Y-%m-%d")
> Date
[1] "1973-12-09" "1974-08-29"


> object <- Date
> c(typeof = typeof(object), mode = mode(object), storag.mode = storage.mode(object),
     class = class(object))
     typeof        mode storag.mode       class
   "double"   "numeric"    "double"      "Date"
```

Note that the storage mode of a date object is `double` and the object itself is an object of class `Date`.
You can add a number to a date object, the number is interpreted as the number of days to add to the date.

```
> Date + 19
[1] "1973-12-28" "1974-09-17"
```

Note that the default formats follow the rules of the ISO 8601 international standard which expresses a day as "2001-02-03".

Difftime objects

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

```
> difftime <- Date[2] - Date[1]
> difftime
Time difference of 263 days


> object <- difftime
> c(typeof = typeof(object), mode = mode(object), storag.mode = storage.mode(object),
     class = class(object))
     typeof        mode storag.mode       class
   "double"   "numeric"    "double"  "difftime"
```

POSIXt objects

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.

```
> posixDate <- as.POSIXct("2003-01-23", tz = "")
> posixDate
[1] "2003-01-23 CET"
```

By default the time zone is unspecified, `tz=""`. If we unclass the object

```
> unclass(posixDate)
[1] 1043276400
attr(,"tzone")
[1] ""
```

we see that the time is measured in a number counted from some origin.

```
> posixDateTime <- as.POSIXct("2003-04-23 15:34")
> posixDateTime
[1] "2003-04-23 15:34:00 CEST"


> object <- posixDate
> c(typeof = typeof(object), mode = mode(object), storag.mode = storage.mode(object),
     class = class(object))
     typeof        mode storag.mode      class1      class2
   "double"   "numeric"    "double"    "POSIXt"   "POSIXct"
```

The function `as.POSIXlt()` converts a date where the atomic parts of the data and time can be retrieved.

```
> ltDateTime <- as.POSIXlt(posixDateTime)
> ltDateTime
[1] "2003-04-23 15:34:00 CEST"
> unclass(ltDateTime)
$sec
[1] 0

$min
[1] 34

$hour
[1] 15

$mday
[1] 23

$mon
[1] 3

$year
[1] 103

$wday
[1] 3

$yday
```

```
[1] 112

$isdst
[1] 1

attr(,"tzone")
[1] ""     "CET"  "CEST"
```

The `strptime()` function

A very useful function is `strptime`; it is used to convert a certain character representation of a date (and time) into another character representation. To do this, you need to provide a conversion specification that starts with a % followed by a single letter.

```
> timestamps <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
> Date <- strptime(timestamps, "%d%b%Y")
> posixDate <- as.POSIXct(Date)
> posixDate
[1] "1960-01-01 CET" "1960-01-02 CET" "1960-03-31 CET" "1960-07-30 CET"


> object <- posixDate
> c(typeof = typeof(object), mode = mode(object), storage.mode = storage.mode(object),
    class = class(object))
      typeof         mode storage.mode       class1       class2
    "double"    "numeric"     "double"     "POSIXt"    "POSIXct"


> 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")
> DateTime <- paste(dates, times)
> DateTimeStamps <- strptime(DateTime, "%m/%d/%y %H:%M:%S")
> posixDateTimeStamps <- as.POSIXct(DateTimeStamps)
```

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.

```
> posixDateTimeStamps + 13
[1] "1992-02-27 23:03:33 CET" "1992-02-27 22:30:09 CET"
[3] "1992-01-14 01:03:43 CET" "1992-02-28 18:21:16 CET"
```

The `difftime()` function

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

```
> posix2 <- as.POSIXct("2004-01-23 14:33")
> posix1 <- as.POSIXct("2003-04-23")
> diffPosix <- posix2 - posix1
> diffPosix
```

```
Time difference of 275.65 days
```

A `difftime` object can also be created using the function `as.difftime`, and you can add a difftime object to a POSIXct object.

```
> posixDateTimeStamps
[1] "1992-02-27 23:03:20 CET" "1992-02-27 22:29:56 CET"
[3] "1992-01-14 01:03:30 CET" "1992-02-28 18:21:03 CET"
> diffPosix
Time difference of 275.65 days
> `+.POSIXt`(posixDateTimeStamps, diffPosix)
[1] "1992-11-29 14:36:20 CET" "1992-11-29 14:02:56 CET"
[3] "1992-10-15 16:36:30 CET" "1992-11-30 09:54:03 CET"
```

To extract the weekday, month or quarter from a POSIXct object use the `R` functions `weekdays()`, `months()` and `quarters()`.

```
> weekdays(posixDateTimeStamps)
[1] "Thursday" "Thursday" "Tuesday"  "Friday"
```

Another handy function is `Sys.time()`, which returns the current date and time.

```
> Sys.time()
[1] "2010-07-20 17:48:38 CEST"
```

There are some `R` packages that can handle dates and time objects. For example, the packages `zoo`, `chron`, `tseries`, `its` and `timeDate`. `timeDate` especially provides a set of powerful functions to maintain and manipulate dates and times.

4.3 INTEGERS

Integers are natural numbers. They are represented as type *Integers*. However, not only integer numbers are represented by this data type, but also factors. These are discussed below.

Natural numbers

Integers can be used to represent counting variables, for example the number of assets in a portfolio.

```
> nAssets <- as.integer(15)
> is.integer(nAssets)
[1] TRUE
```

```
> object <- nAssets
> c(typeof = typeof(object), mode = mode(object), storage.mode = storage.mode(object),
     class = class(object))
      typeof         mode storage.mode        class
   "integer"    "numeric"    "integer"    "integer"
```

Note that 15.0 is not an integer!

```
> nAssets <- 15
> is.integer(nAssets)
[1] FALSE
```

So the number 15 of type `integer` in R is not the same thing as a 15.0 of type 'double'. However, you can mix objects of type 'double' and 'integer' in one calculation without any problems.

```
> nEquities <- as.integer(16)
> nBonds <- as.integer(6)
> percentBonds <- 100 * nBonds/(nEquities + nBonds)
> ans <- round(percentBonds, 1)
> ans
[1] 27.3

> typeof(ans)

[1] "double"
```

The answer is of type double and is 27.3%.

Factors

The factors are used to represent categorical data, i.e. data for which the value range is a collection of codes. For example:

- variable `exchange` with values "NASDAQ", "NYSE" and "AMEX".
- variable `FinCenter` with values: "Europe/Zurich" or "London".

An individual code of the value range is also called a *level* of the factor variable. Therefore, the variable `exchange` is a factor variable with three levels, "NASDAQ", "NYSE" and "AMEX".
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 and of type factor, do the following:

```
> exchange <- c("NASDAQ", "NYSE", "NYSE", "AMEX", "NASDAQ")
```

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

```
> exchange <- factor(exchange)
> exchange
[1] NASDAQ NYSE   NYSE   AMEX   NASDAQ
Levels: AMEX NASDAQ NYSE
```

So what is the object type of factors?

```
> object <- exchange
> c(typeof = typeof(object), mode = mode(object), storag.mode = storage.mode(object),
     class = class(object))
     typeof        mode storag.mode       class
  "integer"   "numeric"   "integer"    "factor"
```

Factors are of type `integer`.
Use the function `levels()` to see the different levels of a factor variable.

```
> levels(exchange)
[1] "AMEX"   "NASDAQ" "NYSE"
```

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

```
> exchange <- c(1, 2, 2, 3, 1)
```

The object `exchange` is an integer variable, so it needs to be transformed to a factor.

```
> exchange <- factor(exchange)
> exchange
[1] 1 2 2 3 1
Levels: 1 2 3
```

The object `exchange` looks like in integer variable, but it is not. The 1 here represents level "1". Therefore arithmetic operations on the variable are not possible:

```
exchange + 4
[1] NA NA NA NA NA
Warning message:
In Ops.factor(exchange, 4) : + not meaningful for factors
```

It is better to rename the levels, so level "1" becomes "AMEX", level "2" becomes "NASDAQ", and level "3" becomes "NYSE":

```
> levels(exchange) <- c("AMEX", "NASDAQ", "NYSE")
> exchange
[1] AMEX   NASDAQ NASDAQ NYSE   AMEX
Levels: AMEX NASDAQ NYSE
```

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

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

The "1" is assigned to the "AMEX" level, only because alphabetically "AMEX" 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:

```
> Position <- c("Long", "Short", "Neutral", "Short", "Neutral",
    "Long", "Short")
> Position <- ordered(Position, levels = c("Short", "Neutral",
    "Low"))
> Position
[1] <NA>    Short   Neutral Short   Neutral <NA>    Short
Levels: Short < Neutral < Low
```

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.

```
> Position.numeric <- as.double(Position)
> Position.numeric
[1] NA  1  2  1  2 NA  1
```

4.4 COMPLEX

Objects of type `complex` are used to represent complex numbers. In statistical data analysis you will get in contact with them for example in the field of spectral analysis of time series. Use the function `as.complex()` or `complex()` to create objects of type `complex`.

```
> cplx1 <- as.complex(-25 + (0+5i))
> sqrt(cplx1)
[1] 0.4975+5.0247i


> cplx2 <- complex(5, real = 2, im = 6)
> cplx2
[1] 2+6i 2+6i 2+6i 2+6i 2+6i


> object <- cplx2
> c(typeof = typeof(object), mode = mode(object), storag.mode = storage.mode(object),
    class = class(object))
     typeof        mode storag.mode       class
  "complex"   "complex"   "complex"   "complex"
```

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

```
> sqrt(as.complex(-1))
[1] 0+1i
```

4.5 LOGICAL

An object of class `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.

```
> test <- (bid > ask)
```

```
> object <- test
> c(typeof = typeof(object), mode = mode(object), storag.mode = storage.mode(object),
     class = class(object))
     typeof        mode storag.mode       class
  "logical"   "logical"   "logical"   "logical"
```

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

```
> is.double(1.0014)
[1] TRUE
> is.double(bid)
[1] TRUE
```

Logical expressions are often built from logical operators:

LISTING 4.3: LOGICAL OPERATORS.

```
Operator:
<           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.

```
> bid != ask
[1] TRUE
```

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. For example is the number of elements in vector larger than a given number?

```
> prices <- c(1.55, 1.61, 1.43, 1.72, 1.69)
> sum(prices > 1.62)
[1] 2
```

4.6 MISSING DATA

We have already seen the symbol `NA`. In R it is used to represent *missing* data (*Not Available*). The type of the NA symbol depends on the initial vector class where the data is missing. If we are working with a vector in double storage mode, the `NA` will also be in double storage mode. Likewise for other classes as `character`, `logical` or `integer`.

```
> (object <- NA)
[1] NA
> c(typeof = typeof(object), mode = mode(object), storag.mode = storage.mode(object),
     class = class(object))
     typeof        mode storag.mode       class
  "logical"   "logical"   "logical"   "logical"


> (object <- as.character(NA))
[1] NA
> c(typeof = typeof(object), mode = mode(object), storag.mode = storage.mode(object),
     class = class(object))
     typeof        mode storag.mode       class
"character" "character" "character" "character"
```

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))
> is.nan(z)
[1] FALSE  TRUE
```

4.7 CHARACTER

A character object is represented by a collection of characters between double or single quotes, " and '. One way to create character objects is as follows.

```
> letters <- c("a", "b", "c")
> letters
[1] "a" "b" "c"
> typeof(letters)
[1] "character"


> exchange <- "Tokyo Stock Exchange"
> exchange
[1] "Tokyo Stock Exchange"


> object <- exchange
> c(typeof = typeof(object), mode = mode(object), storag.mode = storage.mode(object),
     class = class(object))
     typeof        mode storag.mode       class
"character" "character" "character" "character"
```

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

4.8 NULL

In R `NULL` represents the null object and is often returned by expressions and functions whose value is undefined.

```
> object <- NULL
> c(typeof = typeof(object), mode = mode(object), storag.mode = storage.mode(object),
     class = class(object))
     typeof        mode storag.mode       class
     "NULL"      "NULL"      "NULL"      "NULL"
```

PART II

PROGRAMMING

CHAPTER 5

WRITING FUNCTIONS

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.

5.1 WRITING YOUR FIRST FUNCTION

The general form of a function is given by:

```
functionname <- function(arg1, arg2,...) {
    <<expressions>>
}
```

In the above display `arg1` and `arg2` in the function header are input arguments of the function. Note that a function does not 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. Thus, it is not necessary to explicitly use `return()`. The following short function `tmean` calculates the mean of a vector `x` by removing the `k` percent smallest and the `k` percent largest elements of the vector. We call this mean a trimmed mean, therefore we named the function `tmean`

```
> tmean <- 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 run.

```
> test <- rnorm(100)
> tmean(test, 0.05)
[1] 0.047795
```

The function `tmean` calls two standard functions, `quantile` and `mean`. Once `tmean` 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.

Saving your function in a script file

You can use your favourite text editor to create or edit functions. Use the function `source` to evaluate expressions from a file. Suppose *tmean.R* is a text file, saved on your hard disk, containing the function definition of `tmean()`. In this example we use the function `dump()` to export the `tmean()` to a text file.

```
> tmean <- function(x, k) {
     xt <- quantile(x, c(k, 1 - k))
     mean(x[x > xt[1] & x < xt[2]])
 }
> dump("tmean", "tmean.R")
```

You can load the function `tmean` in a new `R` session by using the `source()` function. It is important to specify the relative path to your file if `R` has not been started in the same directory where the source file is. You can use the function `setwd()` to change the working directory of your `R` session or use the GUI menu “Change working directory” if available.

```
> source("tmean.R")
```

Now we can run the function:

```
> tmean(test, 0.05)
[1] 0.047795
```

Using comments

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) {
      sqrt(x)
 }
```

Viewing function code

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 such as: `.C`, `.Internal` or `.Call`.

5.2 ARGUMENTS AND VARIABLES

In this section we explain the difference between required and optional arguments, explain the meaning of the `...` argument, introduce local variables, and show the different options for returning an object from a function.

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
 }
```

Run it

```
> power(5)
[1] 25
```

Bear in mind that x is a required argument. You have to specify it, otherwise you will get an error.

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

To compute the third power of x, we can specify a different value for k and set it to 3:

```
> power(5, k = 3)
[1] 125
```

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.

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


> sinPlot(col = "red")
```

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

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. Note that it is not recommended to use global variables in any R code.
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
> reassign <- function() {
      x <- 3
 }


> reassign()
> x
[1] 0
```

If you want to change the global variable `x` with the return value of the function `reassign`, you must assign the function result to `x`. This overwrites the object `x` with the result of the `reassign` function

```
> x <- reassign()
> x
[1] 3
```

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

```
> execFun <- function(x, fun) {
      fun(x)
 }
```

Try it

```
> Sin <- execFun(pi/3, sin)
> Cos <- execFun(pi/3, cos)
> c(Sin, Cos, Sum = Sin * Sin + Cos * Cos)
                    Sum
0.86603 0.50000 1.00000
```

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

Returning an object

Often the purpose of a function is to do some calculations on input arguments and return the result. As we have already seen in all previous examples, by default the last expression of the function will be returned.

```
> sumSinCos <- function(x, y) {
      Sin <- sin(x)
      Cos <- cos(y)
      Sin + Cos
 }


> sumSinCos(0.2, 1/5)
[1] 1.1787
```

In the above example `Sin + Cos` is returned, whereas the individual objects `Sin` and `Cos` will be lost. You can only return one object. If you want to return more than one object, you can return them in a list where the components of the list are the objects to be returned. For example

```
> sumSinCos <- function(x, y) {
     Sin <- sin(x)
     Cos <- cos(y)
     list(Sin, Cos, Sum = Sin + Cos)
 }

> sumSinCos(0.2, 1/5)
[[1]]
[1] 0.19867

[[2]]
[1] 0.98007

$Sum
[1] 1.1787
```

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:

```
> SinCos <- function(x, y) {
     Sin <- sin(x)
     Cos <- cos(y)
     if (Cos > 0) {
         return(Sin + Cos)
     }
     else {
         return(Sin - Cos)
     }
 }

> SinCos(0.2, 1/5)
[1] 1.1787
> sin(0.2) + cos(1/5)
[1] 1.1787
> sin(0.2) - cos(1/5)
[1] -0.7814
```

5.3 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.

```
> myScope <- 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 `myScope`.
- `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 **?**. With lexical scoping, free variables are first resolved in the environment in which the function was created. The following calls to the function `myScope` shows this rule.
In the first example R tries to find `a1` in the environment where `myScope` was created but there is no object `a1`

```
> myScope(8)
Error in myf(8) : object "a1" not found
```

Now let us define the objects `a1` and `a2` but what value was assigned to a2 in the function `insidef`?

```
> a1 <- 10
> a2 <- 1000
> myScope(8)
[1] 1.3921
```

It took `a2` in `myScope`, so `a2` has the value `9`.

5.4 LAZY EVALUATION

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

```
> myf <- function(x, nc = length(x)) {
 }
```

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)
  }
```

5.5 FLOW CONTROL

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.

Tests with `if()`

The general form of the if construction has the form

```
if(test) {
   <<statements1>>
} else {
   <<statements2>>
}
```

where `test` is a logical expression such as `x < 0` or `x < 0 & x > -8`. R evaluates the logical expression; if it results in `TRUE`, it executes the `true`

`statements`. If the logical expression results in `FALSE`, then it executes the `false statements`. Note that it is not necessary to have the `else` block.
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
```

Tests with `switch()`

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` if `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.
Example:
Choosing between two calculation methods:

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

Looping with `for`

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

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

In the `loop` `some expressions` are evaluated for each element `i` in `for_object`.
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]  1.0000  2.7500  5.0625  7.7969 10.8477 14.1357 17.6018 21.2014 24.9010
[10] 28.6758
```

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

Looping with `while()`

```
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.
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")
  }
```

Looping with `repeat()`

```
repeat
{
   <<commands>>
}
```

In the repeat loop «`commands`» are repeated *infinitely*, so `repeat` loops will have to contain a `break` statement to escape them.

CHAPTER 6

DEBUGGING YOUR R FUNCTIONS

Debugging is the methodical process to find and reduce and eliminate warnings and errors returned from your R functions. These warnings and errors are also called *bugs*, therefore the name *debugging*. The goal of debugging is to run your R function behaved as it was expected.
The following functions will help you to analyze and debug your R functions.

LISTING 6.1: SOME FUNCTIONS THAT WILL HELP YOU TO ANALYZE AND DEBUG YOUR R FUNCTIONS

```
Function:
traceback       prints the call stack of the last uncaught error
warning         generates a warning message
stop            stops execution and executes an error action
debug           sets or unsets the debugging flag on a function
browser         interrupt execution and allows for inspection
```

6.1 THE `traceback()` FUNCTION

The R language provide the user with some tools to track down *unexpected behaviour* 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.

6.2 THE FUNCTION `warning()` AND `stop()`

You, as the writer of a function, can also produce errors and warnings. In addition to putting ordinary print statements such as `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)
  }
> NA
```

```
[1] NA
```

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.

```
> NA
[1] NA
```

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.

6.3 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 as an interactive R session, that means you could enter commands such as:

- `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)`.

6.4 THE FUNCTION `browser()`

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.

CHAPTER 7

EFFICIENT CALCULATIONS

The efficiency of calculations depends on how you perform them.

7.1 VECTORIZED COMPUTATIONS

Vectorized calculations, for example, avoid going through 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()` and `sin()` 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 value 1. You could use the following construction (as in C or Fortran)

```
> 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 0.036382 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 0.0053601 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 function `ifelse()`

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 0.078888 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 -0.012361 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 function `cumsum()`

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:

```
> NA
[1] NA
```

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)
```

7.2 THE FAMILY OF `apply()` FUNCTIONS

The function `apply()`

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)
  [1] -0.10290508 -0.03091177  0.00844846  0.05244420 -0.07729698 -0.05689280
  [7]  0.01775888  0.00110060 -0.09423070 -0.04783159  0.05037437  0.08825522
 [13] -0.21584964 -0.08381933  0.01199495  0.07086870  0.01853074 -0.19313128
 [19] -0.01025858  0.19653877  0.13737483  0.10898164 -0.10227143  0.11240556
 [25] -0.04421831 -0.13400429 -0.00150755 -0.04809952  0.04879253 -0.09653811
 [31]  0.06030712 -0.08889533  0.06689322  0.11601070 -0.12333348 -0.09879102
 [37] -0.13101764  0.08756582 -0.01163899  0.17533528 -0.10447925  0.04053187
 [43] -0.01498529 -0.02541122  0.01506532  0.02163973 -0.09112403 -0.05984587
 [49]  0.17087208  0.04706881  0.17634669 -0.19657742 -0.04957147  0.02823138
 [55] -0.10358411  0.03515878  0.10375764  0.16013378 -0.12467199  0.04024889
 [61]  0.16048015  0.01925856  0.06999432 -0.19301662 -0.00078018 -0.00385603
 [67]  0.21474041  0.18043834 -0.16509658 -0.18291511 -0.18284043 -0.04569351
 [73] -0.05721336  0.21039915 -0.05228664 -0.02735980 -0.00435136  0.15807645
 [79] -0.04412718  0.16106210 -0.08323803 -0.00903785  0.12101442  0.06039896
 [85] -0.07566664  0.11868099 -0.00577876 -0.04535101  0.05035219  0.01778280
 [91]  0.04335710  0.12587539 -0.01911259 -0.07600562 -0.08267036  0.01979584
 [97]  0.09390418 -0.01199051 -0.13337728  0.10949662
```

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 row will be calculated, if one chooses 2 then the mean will be calculated for each column. 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 row in a matrix.

```
> thresh <- function(x, d) {
      sum(x > d)
  }
> M <- matrix(rnorm(10000), ncol = 100)
> apply(M, 1, thresh, 0.6)
  [1] 30 30 34 25 35 26 29 32 20 34 24 33 37 38 23 28 25 24 30 24 24 28 30 28 26
 [26] 25 20 32 21 23 25 32 27 22 28 35 28 24 35 17 32 22 23 29 27 25 23 24 23 21
 [51] 24 29 17 19 25 21 25 35 18 26 28 25 34 27 26 23 24 27 32 30 37 20 32 33 21
 [76] 35 21 20 33 29 31 26 27 24 31 35 23 14 27 25 23 27 35 35 24 28 27 27 25 27
```

Notice that the argument `d` is now passed to `apply()`.

The lapply() and sapply() functions

```
> NA
[1] NA
```

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)
 }
> NA
[1] NA
```

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)
$V1
[1] 1 2

$V2
[1] 1 2 3

$V3
[1] 1 2

$V4
[1] 1

$V5
[1] 1 2
```

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

The function `tapply()`

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` and 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 = TRUE))
```

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
 0.42799  0.36805  0.33787 -0.12530
```

Combining the functions `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)
 }
> NA
[1] NA
```

7.3 THE FUNCTION `by()`

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

```
> NA
[1] NA
```

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

```
> NA
[1] NA
```

7.4 THE FUNCTION outer()

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"
```

3D Plots and the function outer()

The function outer() is a very useful function to create 3-dimensional plot. As an example we show how to create the z-component for a perspective plot.

```
> 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)
```

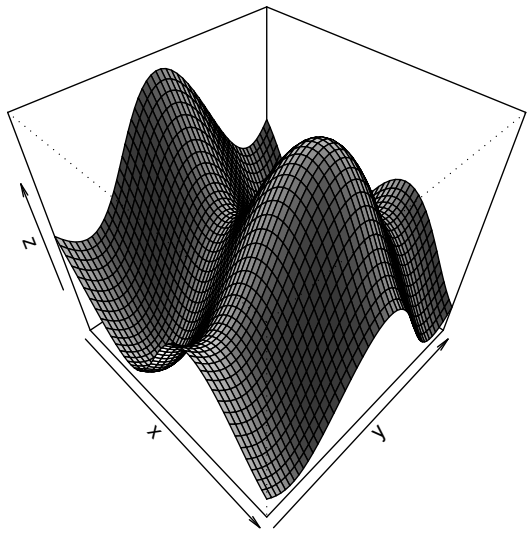

FIGURE 7.1: Perspective plot

CHAPTER 8

USING S3 AND S4 CLASSES

R has two types of object orientations, the older one based on the so called S3 classes, and the newer one, based on the S4 class model.

8.1 S3 CLASS MODEL BASICS

The usual way to define an S3 class in R is simply to attach a "class" attribute to an object. Then define methods (functions) that use this attribute and act properly according to it.

S3 Methods

The function `methods()`

```
> args(methods)
function (generic.function, class)
NULL
```

lists all available methods for an S3 generic function, or all methods for a class.

S3 plot methods

To list the `plot()` methods type

```
> methods(plot)
 [1] plot.acf*           plot.data.frame*    plot.Date*
 [4] plot.decomposed.ts* plot.default        plot.dendrogram*
 [7] plot.density        plot.ecdf           plot.factor*
[10] plot.formula*       plot.hclust*        plot.histogram*
[13] plot.HoltWinters*   plot.isoreg*        plot.lm
[16] plot.medpolish*     plot.mlm            plot.POSIXct*
[19] plot.POSIXlt*       plot.ppr*           plot.prcomp*
[22] plot.princomp*      plot.profile.nls*   plot.spec
[25] plot.spec.coherency plot.spec.phase     plot.stepfun
```

```
[28] plot.stl*           plot.table*         plot.ts
[31] plot.tskernel*      plot.TukeyHSD

   Non-visible functions are asterisked
```

Let us consider some examples. Create a vector of 100 normal random variates and generate an autoregressive process of order 1 and AR coefficient 0.25

```
> set.seed(4711)
> eps = rnorm(120, sd = 0.1)
> y = eps[1]
> for (i in 2:120) y[i] = 0.5 * y[i - 1] + eps[i]
> y = round(y, 3)
> names(y) = paste(100 * rep(1991:2000, each = 12) + rep(1:12,
     times = 10))
> y
 199101 199102 199103 199104 199105 199106 199107 199108 199109 199110 199111
  0.182  0.228  0.234  0.076 -0.023 -0.162  0.001 -0.096 -0.053  0.021 -0.088
 199112 199201 199202 199203 199204 199205 199206 199207 199208 199209 199210
-0.201 -0.097 -0.006  0.017  0.166  0.036  0.056 -0.025  0.090  0.116  0.066
 199211 199212 199301 199302 199303 199304 199305 199306 199307 199308 199309
  0.131  0.039 -0.117 -0.039 -0.141 -0.086 -0.110  0.033  0.038  0.196  0.212
 199310 199311 199312 199401 199402 199403 199404 199405 199406 199407 199408
  0.047 -0.038 -0.095 -0.076 -0.294 -0.331  0.002 -0.126 -0.063 -0.055 -0.222
 199409 199410 199411 199412 199501 199502 199503 199504 199505 199506 199507
-0.051  0.049  0.048  0.219  0.191 -0.076 -0.072  0.018 -0.038 -0.169 -0.138
 199508 199509 199510 199511 199512 199601 199602 199603 199604 199605 199606
-0.160 -0.112 -0.256 -0.055  0.066  0.084  0.121  0.075  0.035  0.013 -0.086
 199607 199608 199609 199610 199611 199612 199701 199702 199703 199704 199705
-0.104 -0.129  0.032  0.016 -0.015  0.012  0.082  0.058  0.037  0.129  0.106
 199706 199707 199708 199709 199710 199711 199712 199801 199802 199803 199804
  0.292  0.339  0.168  0.061 -0.005  0.053  0.135  0.115 -0.147 -0.041  0.005
 199805 199806 199807 199808 199809 199810 199811 199812 199901 199902 199903
  0.057 -0.082 -0.197 -0.119 -0.011  0.012  0.084 -0.044 -0.075 -0.080  0.114
 199904 199905 199906 199907 199908 199909 199910 199911 199912 200001 200002
-0.004 -0.041  0.019  0.088  0.042 -0.173 -0.111 -0.020  0.074  0.108  0.023
 200003 200004 200005 200006 200007 200008 200009 200010 200011 200012
-0.092 -0.006 -0.084 -0.133 -0.050 -0.118 -0.013 -0.057 -0.064  0.022
```

Plot the numeric vector y

```
> x = y
> class(x)
[1] "numeric"
> plot(x)
```

plot y converted in a time series object

```
> x = ts(y, start = c(1991, 1), frequency = 12)
> class(x)
[1] "ts"
> plot(x)
```

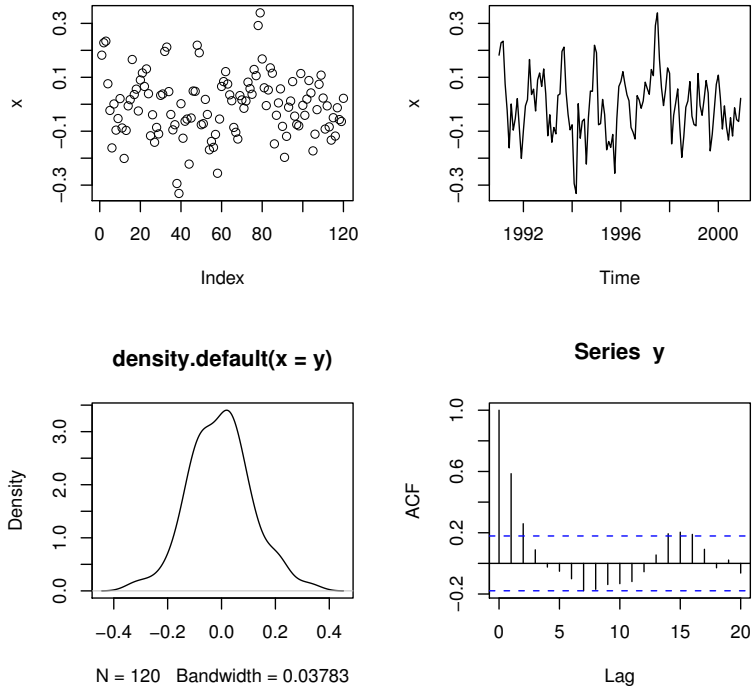

FIGURE 8.1: Vector, time series, density and acf Plots.

plot the density of y obtained from a kernel density estimation

```
> x = density(y)
> class(x)
```

```
[1] "density"
```

```
> plot(x)
```

or plot the auto correlation function of y

```
> x = acf(y, plot = FALSE)
> class(x)
```

```
[1] "acf"
```

```
> plot(x)
```

In all four cases we used the generic function `plot()` to create the graph presented in the figure.

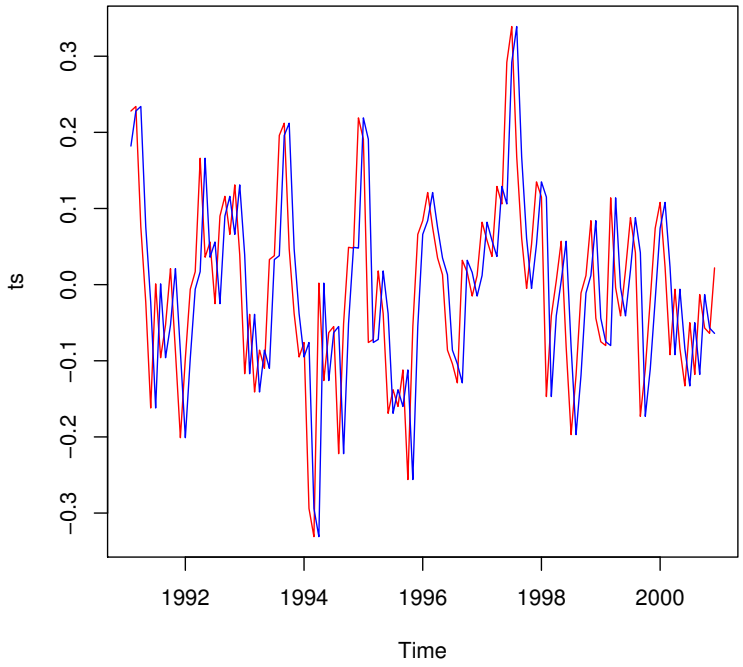

FIGURE 8.2: Lagged Time Series Plot.

Multivariate time series and regression plots

Now let us go one step further and use the function `lm()` to extract the AR coefficient from the autoregressive process `y`. Let us first define a multivariate time series with the original and a time lagged series

```
> x = ts(y, start = c(1991, 1), frequency = 12)
> ts = na.omit(cbind(x, lagged = lag(x, -1)))
> plot(ts, plot.type = "single", col = c("red", "blue"))
```

Now the generic plot function plots both series, the original and the lagged in a singe plot.
In the next step we fit the AR coefficient by linear regression

```
> LM = lm(x ~ lagged, data = ts)
> LM

Call:
lm(formula = x ~ lagged, data = ts)
```

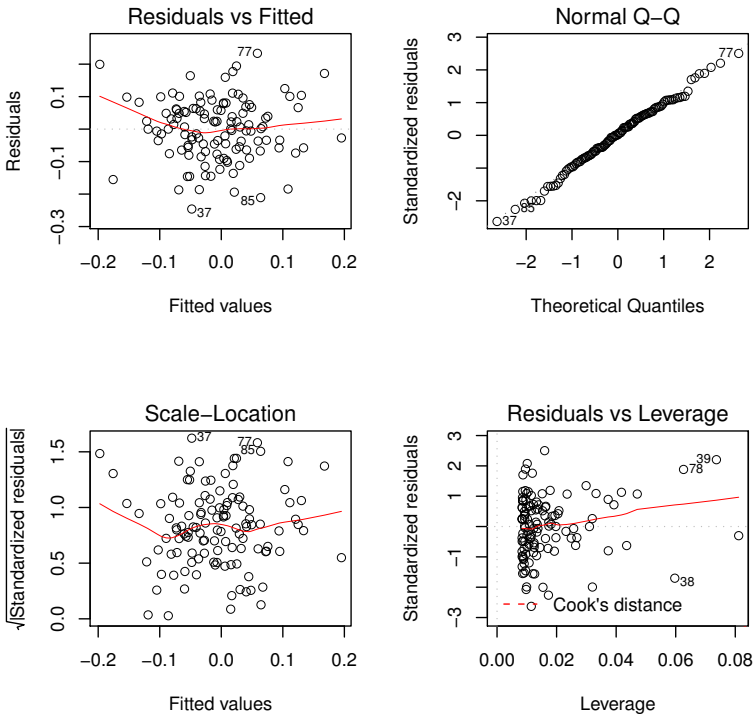

FIGURE 8.3: Linear regression Plot.

```
Coefficients:
(Intercept)       lagged
   -0.00345      0.58597

> class(LM)

[1] "lm"

> par(mfrow = c(2, 2))
> plot(LM)
```

Now the generic function `plot()` produces a figure with four plots, "Residuals vs Fitted", "Norma Q-Q", "Scale-Location", and "Residuals vs Leverage".

Objects of class `"lm"` come with many other generic functions including `print()`, `summary()`, `coef()`, `residuals()`, `fitted()`, or `predict()`. These generic functions have methods for many other objects defined in the field of modeling.

Showing the code of a non-visible method

To display the code of a method, just print the name of the generic function together with the name of the method, e.g. `plot.default()`, `plot.ts()`, or

```
> plot.density
function (x, main = NULL, xlab = NULL, ylab = "Density", type = "l",
    zero.line = TRUE, ...)
{
    if (is.null(xlab))
        xlab <- paste("N =", x$n, "  Bandwidth =", formatC(x$bw))
    if (is.null(main))
        main <- deparse(x$call)
    plot.default(x, main = main, xlab = xlab, ylab = ylab, type = type,
        ...)
    if (zero.line)
        abline(h = 0, lwd = 0.1, col = "gray")
    invisible(NULL)
}
<environment: namespace:stats>
```

However in the case of `plot.acf()` we get an error. The reason is that `plot.acf()` is a non-visible function. Use instead the function call `getAnywhere(plot.acf)` and you will get the code of the function returned.

8.2 S4 CLASS MODEL BASICS

CHAPTER 9

R PACKAGES

R packages, as the name tells us, are a collection of R functions, including help in form of manual pages and vignettes, and if required source code for interfacing Fortran, C or C++ code to R functions. In addition a package also holds a description file for the package and a copyright and license information file. In the following we show how to use packages to extend R's functionality.

9.1 BASE R PACKAGES

"The standard (or base) packages are considered part of the R source code. They contain the basic functions that allow R to work, and the datasets and standard statistical and graphical functions that are described in this manual. They should be automatically available in any R installation.", *from An Introduction to R.*
The R distribution comes with the following packages:

LISTING 9.1: LIST OF BASE PACKAGES

```
base        Base R functions
datasets    Base R datasets
grDevices   Graphics devices for base and grid graphics
graphics    R functions for base graphics
grid        The graphics layout capabilities
methods     Formally defined methods and classes for R objects
splines     Regression spline functions and classes
stats       R statistical functions
stats4      Statistical functions using S4 classes.
tcltk       Interface to Tcl/Tk GUI elements
tools       Tools for package development and administration
utils       R utility functions.
```

9.2 CONTRIBUTED R PACKAGES FROM CRAN

Contributed Packages are written by R developers and can be obtained from the CRAN server. A list of the packages with descriptions is available on the CRAN server from which the packages can also be downloaded.

```
http://cran.r-project.org/web/packages/
```

9.3 R PACKAGES UNDER DEVELOPMENT FROM R-FORGE

Packages under current development can be downloaded from the R-forge server.

```
http://r-forge.r-project.org/
```

9.4 R PACKAGE USAGE

Listing and loading packages

To see which packages are installed, use the command

```
library()
```

To load a particular package, e.g. MASS, use the command

```
library(MASS)
```

To see which packages are currently loaded, use the command

```
search()
```

Installing packages

To install an R package use the command

```
install.packages(pkgname)
```

Alternatively you can call its GUI equivalent. On most systems `install.packages()` with no arguments will allow packages to be selected from a list box.

GUI package menu

The use of packages becomes more convenient using the Packages GUI. The menu points are

LISTING 9.2: GUI PACKAGES MENU.

```
Load Package ...
Set CRAN Mirror ...
Select Repositories ...
```

```
Install Package(s) ...
Update Packages ...
Install Package(s) from local zip files ...
```

9.5 PACKAGE MANAGEMENT FUNCTIONS

The following table lists functions from R's `utils` package which are very helpful to use and manage packages.

LISTING 9.3: PACKAGE MANAGEMENT FUNCTIONS.

```
Function:             Description:
available.packages    Download packages from CRAN-like repositories
compareVersion        Compare Two Package Version Numbers
contrib.url           Download Packages from CRAN-like repositories
download.packages     Download Packages from CRAN-like repositories
INSTALL               Install Add-on Packages
install.packages      Download Packages from CRAN-like repositories
installed.packages    Find Installed Packages
new.packages          Download Packages from CRAN-like repositories
packageDescription    Package Description
packageStatus         Package Management Tools
setRepositories       Select Package Repositories
update.packages       Download Packages from CRAN-like repositories
update.packageStatus  Package Management Tools
upgrade.packageStatus Package Management Tools
```

PART III

PLOTTING

CHAPTER 10

HIGH LEVEL PLOTS

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 **?**. In this chapter we will first discuss the graphical functions that can be found in the base R system and the lattice package.
The graphical functions in the base R system, can be divided into two groups: (i) high level plot functions, and (ii) low level plot functions
High level plot functions produce 'complete' graphics and will erase existing plots if not specified otherwise. Low level plot functions are used to add graphical objects like lines, points and texts to existing plots.

10.1 SCATTER PLOTS

The most elementary plot function is `plot`. In its simplest form it creates a scatterplot of two input vectors. Let us consider an integer index ranging from 1 to 260 and an artificial log-return series of the same length simulated from normal random variates. Note that the cumulative values form a random walk process.[1]

```
> set.seed(4711)
> Index <- 1:260
> Returns <- rnorm(260)
> plot(x = Index, y = Returns)
```

To create a plot with a title use the argument main in the plot function.

```
> plot(x = Index, y = Returns, main = "Artificial Returns")
```

[1] Note that he setting of a random number seed by the function `set.seed()` lets you simulate exactly the same set of random numbers as we used in the example.

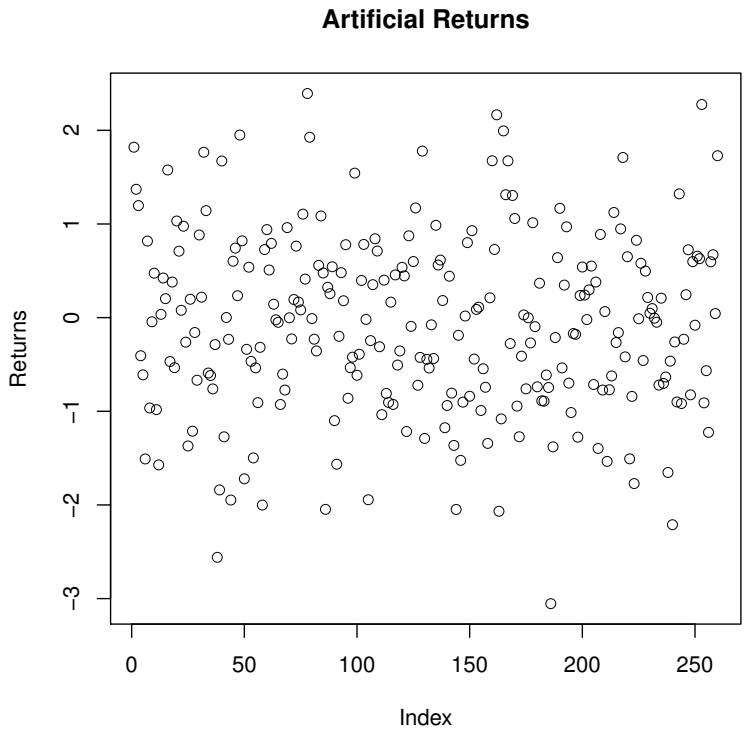

FIGURE 10.1: A scatterplot of 260 artificial returns plotted agains the index ranging from 1 to 260. The length of the series is approximately that of an one year with records recorded every business day. A title is added to the plot.

10.2 LINE PLOTS

Several flavours of simple line plots can be drawn using R's base function `plot()` selecting the appropriate `type` argument for a desired line style.

A simple line plot

For a simple line plot which connects consecutive points we have to set the option `type = "l"` in the `plot()` function. Let us return to the previous example and let us replot the data points in a different way.

```
> par(mfrow = c(2, 1))
> Price = cumsum(Returns)
> plot(Index, Price, type = "l", main = "Line Plot")
> plot(Index, Returns, type = "h", main = "Histogram-like Vertical Lines")
```

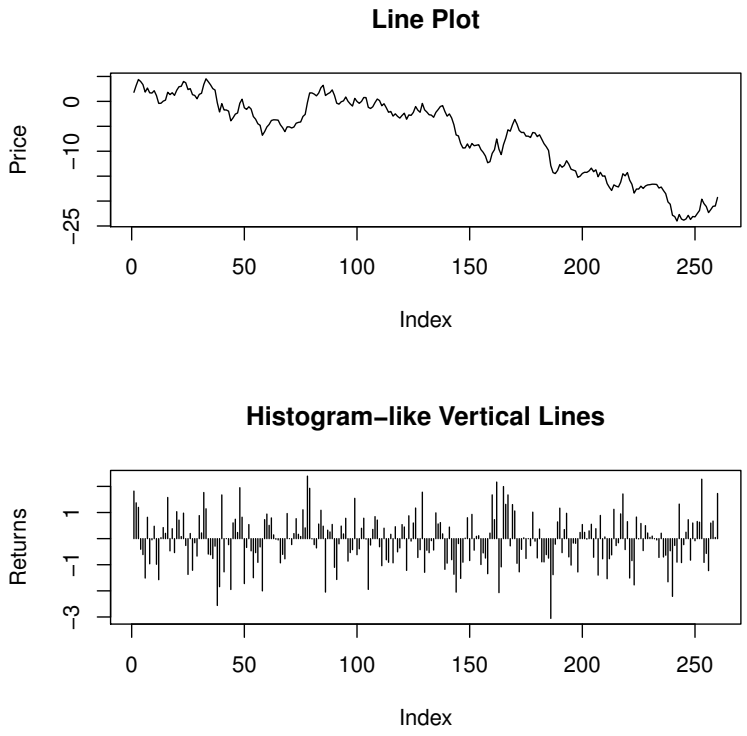

FIGURE 10.2: Two flavours of series plot. The upper graph shows a line plot of cumulated returns created with `type="l"`, and the lower graph shows a plot of returns by histogram like vertical lines created with `type="h"`.

Here the `Returns` are the series of 260 artificially created log-returns standardized with mean zero and variance one. Capital `Price` is the the cumulated log-return series describing a driftless random walk which can also be interpreted as an artificial created index series.

Drawing functions or expressions

In case of drawing functions or expressions, the function `curve` can be handy, it takes some work away to produce a line graph. We show this displaying the density of the log-returns underlying a histogram plot.

```
> hist(Returns, probability = TRUE)
> curve(dnorm(x), min(Returns), max(Returns), add = TRUE, lwd = 2)
```

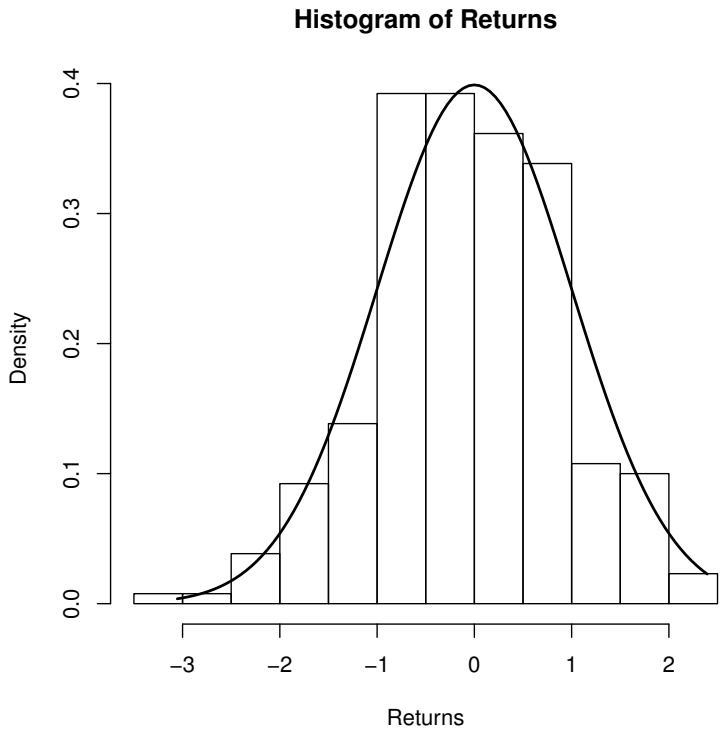

FIGURE 10.3: A histogram plot of the log-returns overlayed by their density function. The density line was created by the `curve()` function.

10.3 MORE ABOUT THE `plot()` FUNCTION

The `plot()` function is very versatile function. The `plot()` function is a so called generic function. Depending on the class of the input object the function will call a specific plot method. Some examples:

LISTING 10.1: GRAPHS PRODUCED BY THE GENERIC PLOT FUNCTION `plot()`

```
Function:
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(ts)                creates a time series plot if ts is a 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; xpos is a POSIXct object and y a numeric vector
plot(f, low, up)	creates a graph of the function f between low and up

The code below shows four examples of the different uses of the function `plot()`.

```
> factorData <- factor(sample(c(rep("AMEX", times = 40), rep("NASDAQ",
      times = 180), rep("NYSE", times = 90))))


> tsData = ts(matrix(rnorm(64), 64, 1), start = c(2001, 1), frequency = 12)


> plot(factorData, col = "steelblue")
> plot(factorData, rnorm(length(factorData)), col = "orange")
> plot(dnorm, min(tsData[, 1]), max(tsData[, 1]), xlab = "Returns",
      yla = "Density", main = "Density")
> grid()
> plot(tsData, xlab = "Index", ylab = "Returns", main = "Return Series")
> abline(h = 0)
```

10.4 DISTRIBUTION PLOTS

R has a variety of plot functions to display the distribution of a data vector. The following function calls can be used to analyze the distribution of log-returns graphically.

LISTING 10.2: PLOTTING FUNCTIONS TO ANALYZE AND DISPLAY FINANCIAL RETURN DISTRIBUTIONS

Function:	
hist	creates a histogram plot
truehist	plots a true histogram with density of total area one
density	computes and displays kernel density estimates
boxplot	produces a box-and-whisker plot
qqnorm	creates a normal quantile-quantile plot
qqline	adds a line through the first and third quartiles
qqplot	produces a QQ plot of two datasets

Note that the above functions can take several arguments for fine tuning the graph, for details we refer to the corresponding help files.

Example: Distribution of USD/EUR FX returns

In the following example we show how to plot the daily log-returns of the USD/EUR foreign exchange rate.

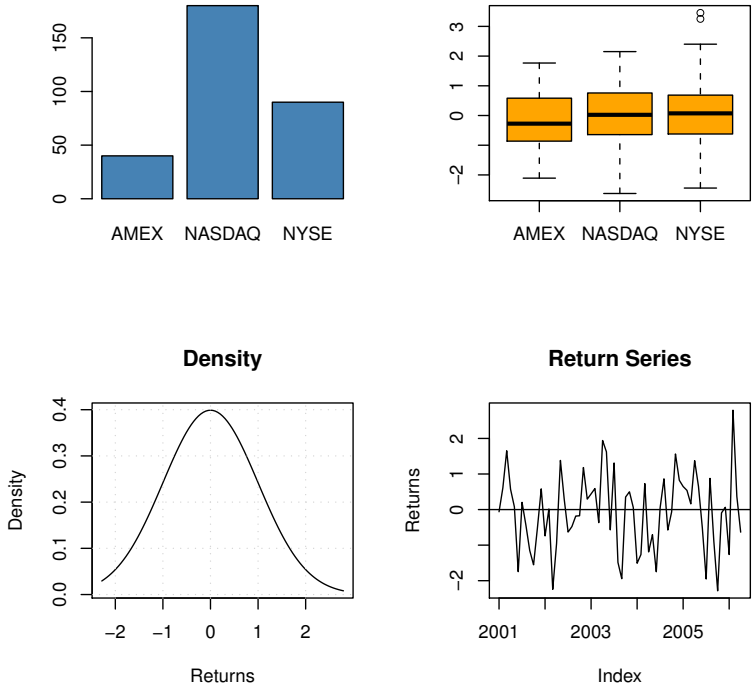

FIGURE 10.4: Different uses of the function plot. Factor plot, box plot, dnsity plot, and time series plot.

The foreign exchange rates can be downloaded from the FRED2 database of the US Federal Reserve Bank in St. Louis.

```
> RATE <- "DEXUSEU"
> URL <- paste("http://research.stlouisfed.org/fred2/series/",
     RATE, "/", "downloaddata/", RATE, ".csv", sep = "")
> URL
[1] "http://research.stlouisfed.org/fred2/series/DEXUSEU/downloaddata/DEXUSEU.csv"

> USDEUR <- read.csv(URL)
> head(USDEUR)

        DATE  VALUE
1 1999-01-04 1.1812
2 1999-01-05 1.1760
3 1999-01-06 1.1636
4 1999-01-07 1.1672
5 1999-01-08 1.1554
6 1999-01-11 1.1534
```

The daily log-return vector is computed from the differences of the logarithms of the rates

```
> USDEUR.RET = diff(log(USDEUR[, 2]))
```

Example: A histogram plot

The generic function `hist()` computes a histogram of the given data values. If the argument `plot=TRUE`, the resulting object of class `"histogram"` is plotted by the function `plot.histogram()`, before it is returned.

```
> hist(USDEUR.RET, col = 2, main = "Histogram Plot")
```

Example: A kernel density estimate

The generic function `density()` computes kernel density estimates. Its default method does so with the given kernel and bandwidth for univariate observations.
The algorithm used in `density.default()` disperses the mass of the empirical distribution function over a regular grid of at least 512 points and then uses the fast Fourier transform to convolve this approximation with a discretized version of the kernel and then uses linear approximation to evaluate the density at the specified points.
Note that there is a generic `plot()` function to plot the density.

```
> density = density(USDEUR.RET)
> plot(density, main = "Kernel Density Estimate")
```

Example: A quantile-quantile plot

`qqnorm()` is a generic function the default method of which produces a normal quantile-quantile plot of the values. The function `qqline()` adds a line to a normal quantile-quantile plot which passes through the first and third quartiles.

```
> qqnorm(USDEUR.RET, pch = 19)
> qqline(USDEUR.RET)
```

Example: A box-and-whisker plot

The function `boxplot()` produces box-and-whisker plot(s) of the given (grouped) values.

```
> boxplot(USDEUR.RET, col = "green", main = "Box-Plot")
```

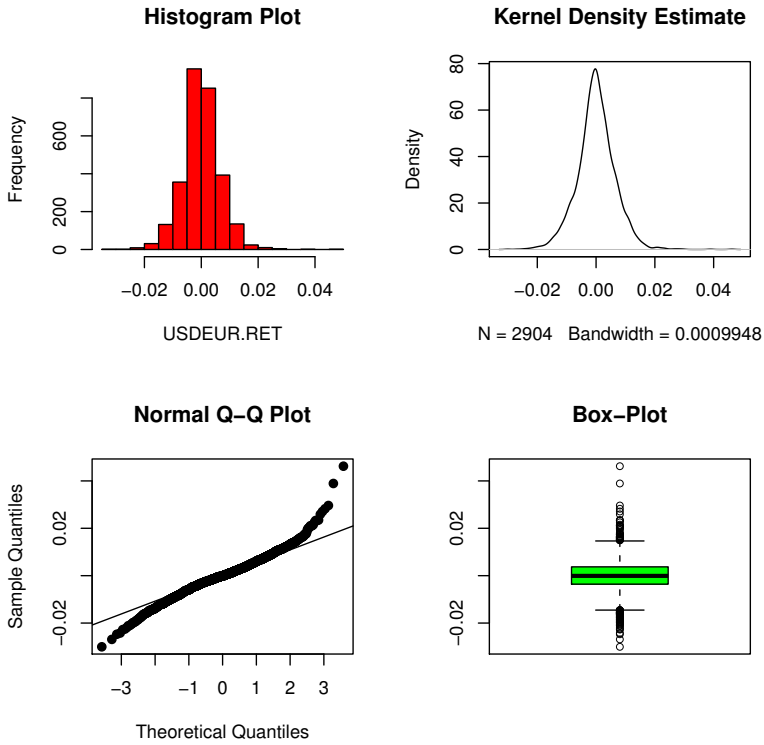

FIGURE 10.5: Different views on distribution of the daily log-returns of the US Dollar / Euro foreign exchange rate.

10.5 PIE AND BAR PLOTS

The functions `pie()` and `barplot()` can be used to draw pie and bar plots.

Data Set - Asset weights of a portfolio

Let us consider a portfolio with the following assets weights

```
> portfolioWeights = c(SwissBonds = 35, SwissEquities = 20, ForeignBonds = 25,
      ForeignEquities = 10, Commodities = 5, PrivateEquities = 5)
```

Let us create plots allowing for different views on the composition of the portfolio.

Example: A vertical bar plot

The following plot command creates a vertical view on the weights bars.

```
> barplot(sort(portfolioWeights), las = 3, col = heat.colors(6),
     offset = 0)
> title(main = "Portfolio Weights")
```

Example: A horizontal bar plot

A horizontal view can be created by the following R command.

```
> barplot(sort(portfolioWeights), horiz = TRUE, las = 1, col = heat.colors(6),
     space = 1)
> text(33.3, 1.5, "%", cex = 1.8)
```

Example: A pie plot

And the last example for the weights plot show how to create a pie plot.

```
> pie(sort(portfolioWeights), init.angle = 20, col = heat.colors(6))
> abline(h = -1)
> mtext(side = 1, line = -0.2, "Portfolio Weights", cex = 0.9,
     adj = 0)
```

Example: Plotting major stock market capitalizations

Data Set - Major stock market capitalizations

The example data set `Capitalization` lists the stock market capitalizations in USD of the worlds's major stock markets for the years 2003 to 2008.
That the data records fit in one printed line, we display the capitalization in units of 1000.

```
> Cap = floor(Capitalization/1000)
> Cap
                2003  2004 2005  2006  2007 2008
Euronext US     1328 12707 3632 15421 15650 9208
TSX Group        888  1177 1482  1700  2186 1033
Australian SE    585   776  804  1095  1298  683
Bombay SE        278   386  553   818  1819  647
Hong Kong SE     714   861 1054  1714  2654 1328
NSE India        252   363  515   774  1660  600
Shanghai SE      360   314  286   917  3694 1425
Tokyo SE        2953  3557 4572  4614  4330 3115
BME Spanish SE   726   940  959  1322  1781  948
Deutsche Boerse 1079  1194 1221  1637  2105 1110
London SE       2460  2865 3058  3794  3851 1868
Euronext EU     2076  2441 2706  3712  4222 2101
SIX SE           727   826  935  1212  1271  857
```

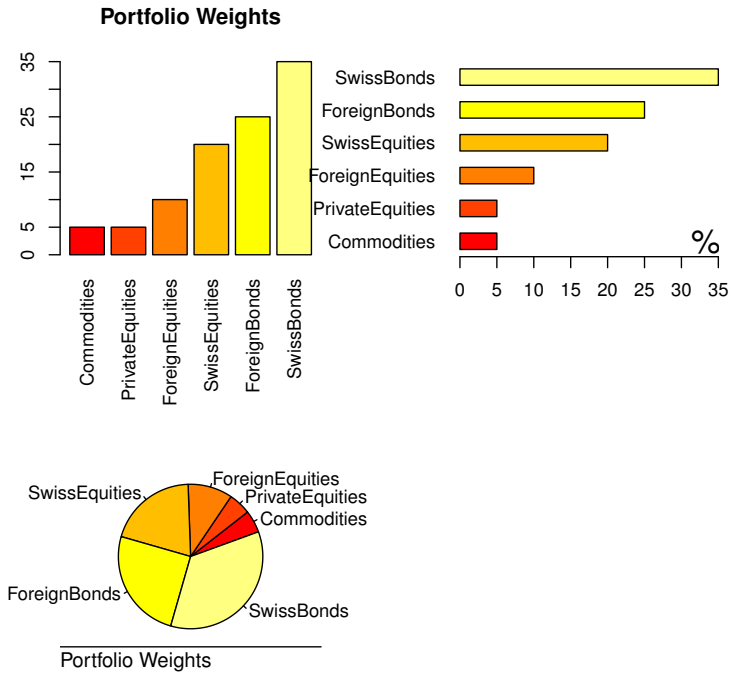

FIGURE 10.6: Different views on the composition of a portfolio The weights of each asset class are printed a s pies or vertical and horizontal bars.

Example: Bar plots of stock market capitalizations

Now we want to express the information graphically in form of a vertical and horizontal bar plot and in form of a pie plot

```
> barplot(t(Cap)/1e+06, beside = TRUE, las = 2, ylab = "Capitalization [Mio USD]")
> title(main = "Major Stock Markets")
> mtext(side = 3, "2003 - 2008")
> barplot(Cap/1e+06, beside = TRUE, ylab = "Capitalization [Mio USD]")
```

10.6 STARS- AND SEGMENTS PLOTS

Stars and segments plots consist of a sequence of equi-angular spokes, called radii, with each spoke representing one of the variables. The data length of a spoke or segment is a measure to the magnitude of the variable which gives the plot a star-like appearance.

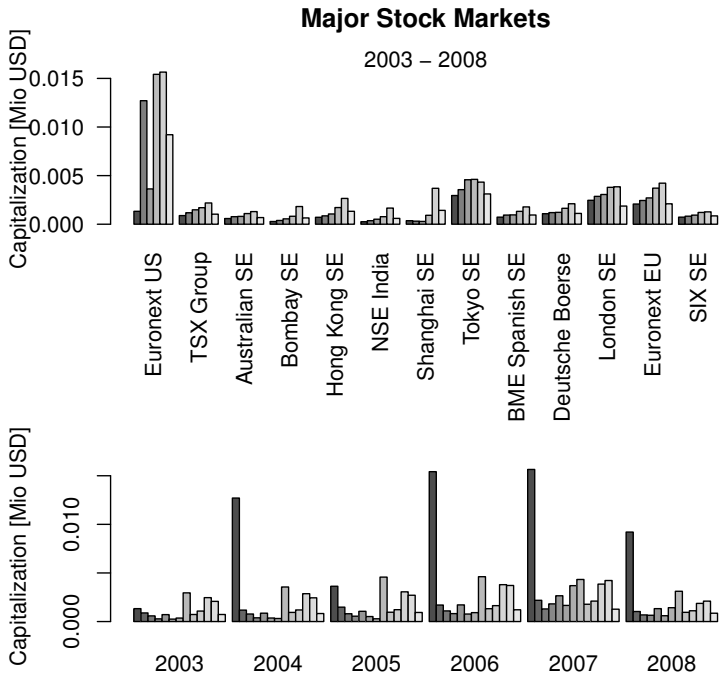

FIGURE 10.7: Matrix Bar Plot of stock market capitalization.

Using the data from the previous section we can draw a graph which shows the growth of the stock markets over the last 5 years and the decline in 2008. Note that the growth of the markets is shown on a logarithmic scale.

```
> palette(rainbow(13, s = 0.6, v = 0.75))
> stars(t(log(Cap)), draw.segments = TRUE, ncol = 3, nrow = 2,
      key.loc = c(4.6, -0.5), mar = c(15, 0, 0, 0))
> mtext(side = 3, line = 2.2, text = "Growth and Decline of Major Stock Markets",
      cex = 1.5, font = 2)
> abline(h = 0.9)
```

Note that to find a nice placement of the stars or segments one has sometimes to fiddle around a little bit with the positioning arguments of the function `stars()`.

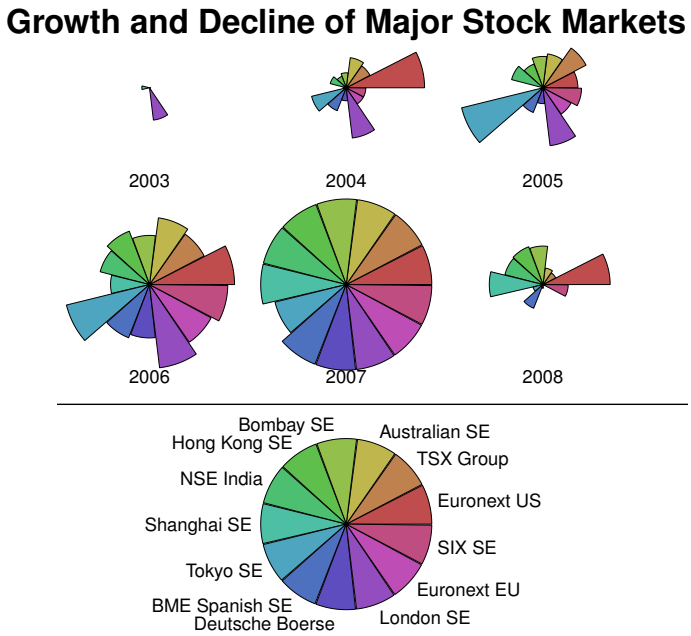

FIGURE 10.8: Growth and Decline of Major Stock Markets

10.7 BI- AND MULTIVARIATE PLOTS

When you have two or more variables (in a data frame) you can use the following functions to display their relationship.

LISTING 10.3: A LIST OF PLOTTING FUNCTIONS TO ANALYZE BI AND MULTIVARIATE DATA SETS

```
Function:
pairs               for data frames the function plots each column
                    against each other
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             creates a contour plot or adds contour lines to
                    an existing plot
filled.contour      produces a contour plot with the areas between
                    the contours filled in solid color
```

image	produces an image plot based on a grid of colored or gray-scale rectangles with colors corresponding to the values of the third variable
persp	draws perspective plots of surfaces over the xÃśy plane.

The code below demonstrates some of the above functions. First let us define some data and set the layout

```
> x <- y <- seq(-4 * pi, 4 * pi, length = 27)
> r <- sqrt(outer(x^2, y^2, "+"))
> z <- cos(r^2) * exp(-r/6)
```

Example: An image plot

The following example shows how to produce an image plot.

```
> image(z, axes = FALSE, main = "Math can be beautiful ...", xlab = expression(cos(r^2) *
      e^{
          -r/6
      }))
```

Example: A dot chart

Now we create a dot chart using the Virginia death rates provided by R's demo data set `VADeaths`

```
> VADeaths
      Rural Male Rural Female Urban Male Urban Female
50-54       11.7          8.7       15.4          8.4
55-59       18.1         11.7       24.3         13.6
60-64       26.9         20.3       37.0         19.3
65-69       41.0         30.9       54.6         35.1
70-74       66.0         54.3       71.1         50.0
```

```
> dotchart(t(VADeaths), xlim = c(0, 100), cex = 0.6)
> title(main = "Insurance - Death Rates in VA")
```

Example: A symbols plot

Next, we plot *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)
> title(main = "Airquality Data")
```

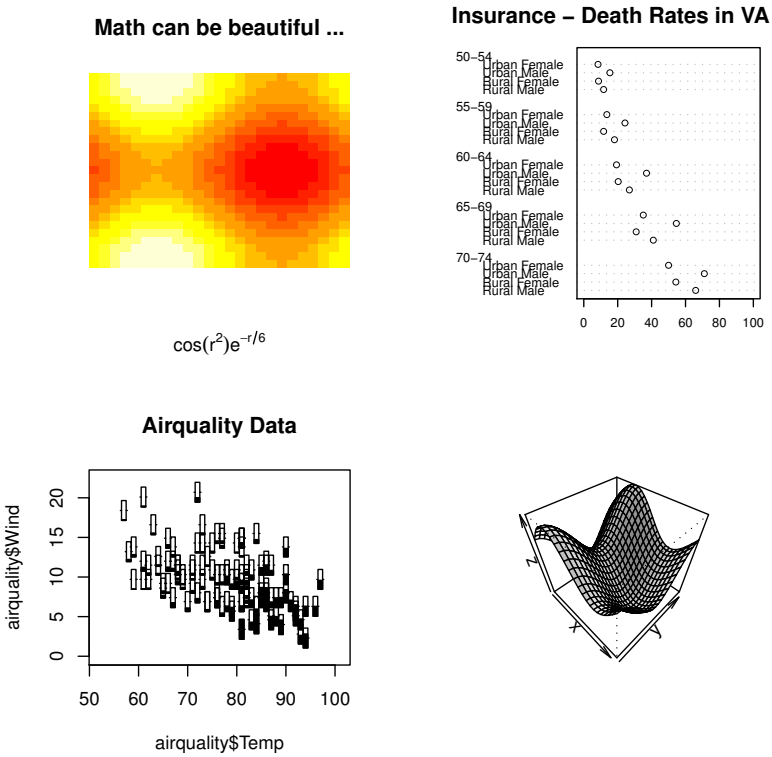

FIGURE 10.9: Example Plots.

Example: A perspective plot

Finally we create a perspective plot

```
> 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)
```

CHAPTER 11

CUSTOMIZING PLOTS

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 already some in the previous section.

11.1 MORE ABOUT PLOT FUNCTION ARGUMENTS

Here are some of the arguments you might want to specify for plots.

LISTING 11.1: SELECTED ARGUMENTS FOR PLOT FUNCTIONS

```
Plot Arguments:
type                what type of plot should be created?
axes                draw or suppress to plot the axes
ann                 draw or suppress to add title and axis labels
pch                 select the type of plotting symbol
cex                 select the size of plotting symbol and text
xlab, ylab          names of the labels for the x and y axes
main                the (main) title of the plot
xlim, ylim          the range of the x and y axes
log                 names of the axes which are to be logarithmic
col, bg             select colour of lines, symbols, background
lty, lwd            select line type, line width
las                 select orientation of the text of axis labels
```

Notice that some of the relevant parameters are documented in `help(plot)` or `plot.default()`, but many only in `help(par)`. The function `par()` is for setting or querying the values of graphical parameters in traditional `R` graphics.

How to modify the plot type

Settings for the plot `type` can be modified using the following identifiers:

LISTING 11.2: TYPE ARGUMENT SPECIFICATIONS FOR PLOT FUNCTIONS

```
Plot Argument:
type                specifies the type of plot
                    "p"        point plot (default)
                    "l"        line plot
                    "b"        both points and lines
                    "o"        overplotted points and lines
                    "h"        histogram like
                    "s"        steps
                    "n"        no plotting
```

Note that by default, the `type` argument is set to `"p"`. If you want to draw the axes first and add points, lines and other graphical elements later, you should use `type="n"`.

How to select a font

With the `font` argument, an integer in the range from 1 to 5, we can select the type of fonts:

LISTING 11.3: FONT ARGUMENTS FOR PLOT FUNCTIONS

```
Plot Arguments:
font                integer specifying which font to use for text
font.axis           font number to be used for axis annotation
font.lab            font number to be used for x and y labels
font.main           font number to be used for plot main titles
font.sub            font number to be used for plot sub-titles
```

If possible, device drivers arrange so that 1 corresponds to plain text (the default), 2 to bold face, 3 to italic and 4 to bold italic. Also, font 5 is expected to be the symbol font, in Adobe symbol encoding.

How to modify the size of fonts

With the argument `cex`, a numeric value which represents a multiplier, we can modify the size of fonts

LISTING 11.4: CEX ARGUMENTS FOR PLOT FUNCTIONS

```
Plot Arguments:
cex                 magnification of fonts/symbols relative to default
cex.axis            magnification for axis annotation relative to cex
cex.lab             magnification for x and y labels relative to cex
cex.main            magnification for main titles relative to cex
cex.sub             magnification for sub-titles relative to cex
```

How to orient axis labels

The argument `las`, an integer value ranging from 0 to 3, allows us to determine the orientation of the axis labels

LISTING 11.5: CEX PARAMETERS FOR PLOT FUNCTIONS

```
Plot Argument:
las                 orientation
  0                   always parallel to the axis [default]
  1                   always horizontal
  2                   always perpendicular to the axis
  3                   always vertical
```

Note that other string/character rotation (via argument `srt` to `par`) does not affect the axis labels.

How to select the line type

The argument `lty` sets the line type. Line types can either be specified as an integer, or as one of the character strings `"blank"`, `"solid"`, `"dashed"`, `"dotted"`, `"dotdash"`, `"longdash"`, or `"twodash"`, where `"blank"` uses invisible lines, i.e. does not draw them.

LISTING 11.6: LTY ARGUMENT FOR PLOT FUNCTIONS

```
Plot Argument:
lty                 sets line type to
  0                   blank
  1                   solid (default)
  2                   dashed
  3                   dotted
  4                   dotdash
  5                   longdash
  6                   twodash
```

Example: Swiss economic data: real GDP vs. population

At the end of this section we present a graph, which shows the usage of some of the arguments presented above. The example shows the correlation between the growth of the Real GDP of Switzerland and the growth of its population.

```
> library(fEcofin)
> swissEconomy

Time Series:
Start = 1964
```

```
End = 1999
Frequency = 1
       GDPR   EXPO   IMPO INTR INFL UNEM POPU
1964  50.74  50.74  56.69 3.97 3.15 0.01 5.79
1965  53.58  53.58  55.19 3.95 3.33 0.01 5.86
1966  56.21  56.21  56.09 4.16 4.84 0.01 5.92
1967  57.42  57.42  56.44 4.61 3.85 0.01 5.99
1968  62.98  62.98  59.78 4.37 2.47 0.01 6.07
1969  70.50  70.50  67.84 4.90 2.41 0.01 6.14
1970  75.18  75.18  79.13 5.82 3.76 0.00 6.19
1971  74.36  74.36  78.10 5.27 6.58 0.00 6.23
1972  75.50  75.50  77.77 4.97 6.60 0.00 6.39
1973  78.52  78.52  81.40 5.60 8.78 0.00 6.43
1974  83.90  83.90  88.99 7.15 9.72 0.01 6.44
1975  75.17  75.17  68.39 6.44 6.69 0.33 6.41
1976  79.19  79.19  71.46 4.99 1.72 0.68 6.35
1977  88.53  88.53  82.37 4.05 1.23 0.39 6.33
1978  85.11  85.11  79.25 3.33 1.07 0.34 6.34
1979  87.87  87.87  89.22 3.45 3.77 0.33 6.36
1980  89.97  89.97  98.61 4.77 3.92 0.20 6.32
1981  93.95  93.95  96.42 5.57 6.56 0.18 6.35
1982  88.55  88.55  87.41 4.83 5.64 0.40 6.39
1983  88.97  88.97  89.09 4.51 2.98 0.80 6.42
1984  96.50  96.50  97.23 4.70 2.89 1.10 6.44
1985 104.19 104.19 103.08 4.78 3.40 1.00 6.47
1986 101.16 101.16  98.05 4.29 0.79 0.80 6.50
1987 100.08 100.08  97.75 4.12 1.46 0.80 6.55
1988 105.34 105.34 104.05 4.15 1.88 0.70 6.59
1989 115.24 115.24 115.84 5.20 3.15 0.60 6.65
1990 115.05 115.05 113.42 6.68 5.37 0.50 6.71
1991 110.11 110.11 105.78 6.35 5.80 1.10 6.80
1992 112.21 112.21 101.21 5.48 4.06 2.50 6.88
1993 112.07 112.07  96.64 4.05 3.36 4.50 6.94
1994 111.72 111.72  97.66 5.23 0.79 4.70 6.99
1995 110.99 110.99  98.30 3.73 1.83 4.20 7.04
1996 113.96 113.96 100.76 3.63 0.86 4.70 7.07
1997 128.35 128.35 114.06 3.08 0.42 5.20 7.09
1998 131.57 131.57 119.09 2.39 0.02 3.90 7.11
1999 139.15 139.15 123.79 3.51 0.89 2.70 7.13
```

Now let us plot the growth in the Swiss Real GDP versus the growth of the population:

```
> plot(x = swissEconomy[, "GDPR"], y = swissEconomy[, "POPU"],
     xlab = "GDP REAL", ylab = "POPULATION", main = "POPULATIN ~ GDP REAL",
     pch = 19, col = "orange", las = 2, font.main = 3, )
```

11.2 GRAPHICAL PARAMETERS

The function `par()` can be used to set or query graphical parameters. Parameters can be set by specifying them as arguments to `par()` in `tag = value` form, or by passing them as a list of tagged values. A call to the function `par()` has the following form

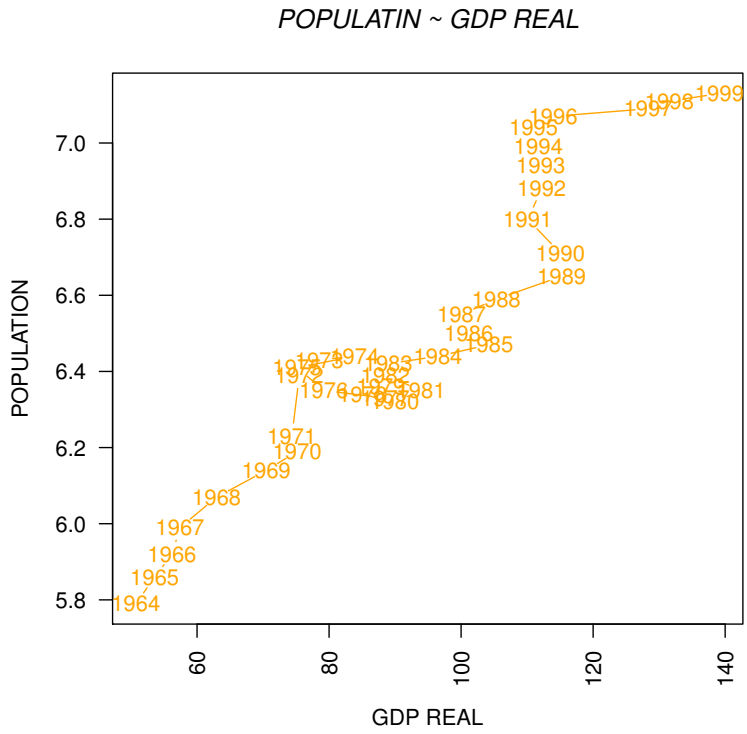

FIGURE 11.1: Population versus GDP Real Plot.

```
par(tag1 = value1, tag2 = value2, ...)
```

In the above code the graphical parameter `tag1` is set to `value1`, graphical parameter `tag2` 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. Show the first ten parameters.

```
> unlist(par()[1:10])
```

```
        xlog         ylog          adj          ann          ask
     "FALSE"      "FALSE"        "0.5"       "TRUE"      "FALSE"
          bg          bty          cex     cex.axis     cex.lab
"transparent"          "o"          "1"          "1"          "1"
```

Once you set a graphical parameter with the `par()` function, that graphical parameter will keep its value until you (i) set the graphical parameter to

another value with the `par()` function, or you (ii) close the graph. R will use the default settings when you create a new plot.
There are several parameters which can only be set by a call to `par()`

```
    "ask",
    "fig", "fin",
    "lheight",
    "mai", "mar", "mex", "mfcol", "mfrow", "mfg",
    "new",
    "oma", "omd", "omi",
    "pin", "plt", "ps", "pty",
    "usr",
    "xlog", "ylog"
```

LISTING 11.7: A LIST OF ARGUMENTS USED FOR THE PAR FUNCTION

```
par Argument:
adj         Determines the way in which text strings are justified
ann         Determines if a plot should be annotaded with titles or not
ask         Determines if the user will be asked interactively for input
bg          The color to be used for the background of the device region
bty         Determines the type of box which is drawn about plotss
cex.*       Gives amount by which text and symbols should be magnified
col.*       A specification for the default plotting colors
crt         A value specifying how single characters should be rotated
family      The name of a font family for drawing text
fg          The color to be used for the foreground of plots
fig         A vector which gives the coordinates of the figure region
fin         The figure region dimensions, width and height
font.*      Specifies which font to use for text
lab         Modifies the default way that axes are annotated.
las         Determines the style of axis labels.
lend        Determines the line end style
lheight     The line height multiplier
ljoin       The line join style
lmitre      The line mitre limit
lty         The line type
lwd         The line width
mai         Gives the margin size
mar         Gives the number of lines of margin on the four sides
mex         Character expansion factor to describe coordinates in the
     margins
mfcol       Determines how subsequent figures will be drawn on one page
mfrow       Determines how subsequent figures will be drawn on one page
mfg         Indicates which figure in an array of figures is to be drawn
     next
mgp         Margin linefor the axis title, labels and axis line
mkh         Height of symbols when the value of pch is an integer
new         Should the next plotting command clean the frame before drawing
oma         Gives the size of the outer margins in lines of text.
omd         Gives the region inside outer margins
```

omi	Gives the size of the outer margins
pch	Symbol or a single character to be used plotting points
pin	The current plot dimensions
plt	Gives the plot region as fraction of the current figure region
ps	The point size of text (but not symbols)
pty	Specifies the type of plot region to be used
srt	The string rotation in degrees
tck	The length of tick marks
tcl	Tick marks as a fraction of the height of a line of text
usr	The extremes of the user coordinates of the plotting region
xaxp	The coordinates of the extreme tick marks
xaxs	Style of axis interval calculation to be used for the x-axis
xaxt	Specifies the x axis type
xlog	Determines the use of a logarithmic x-scale
xpd	Should plotting be clipped to the plot region
yaxp	The coordinates of the extreme tick marks
yaxs	The style of axis interval calculation
yaxt	Specifies the y axis type
ylog	Determines the use of a logarithmic y-scale

For the values of the arguments we refer to the `par()` help page.

11.3 MARGINS, PLOT AND FIGURE REGIONS

A graph consists of three *regions*. A *plot region* surrounded by a *figure region* that is in turn surrounded by four outer margins. The top, left, bottom and right margins. See figure **??**. Usually the high level plot functions create points and lines in the plot region.

Outer margins

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. Show the default parameters for `mar` and `oma`, change these values, and finally reset them.

```
> Par = par(c("mar", "oma"))
> par(oma = c(1, 1, 1, 1))
> par(mar = c(2.5, 2.1, 2.1, 1))
> plot(rnorm(100))
> par(oma = Par$oma)
> par(mar = Par$mar)
```

Multiple plots on one page: `mfrow` and `mfcol`

Use the parameter `mfrow` or `mfcol` to create multiple graphs on one layout. Both parameters are set as follows:

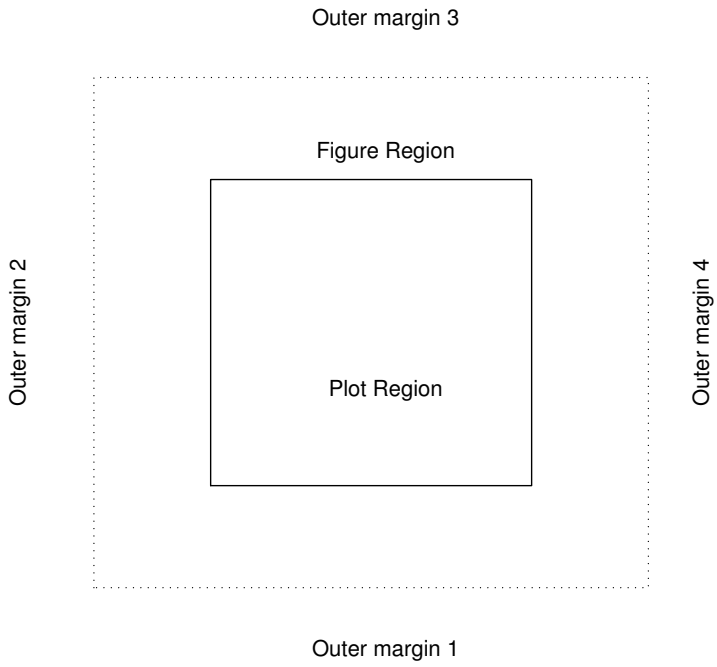

FIGURE 11.2: The different regions of a plot: Outer margins, figure region and plot region.

```
> r = 2
> k = 3
> 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.

Multiple plots on one page: `layout()`

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.

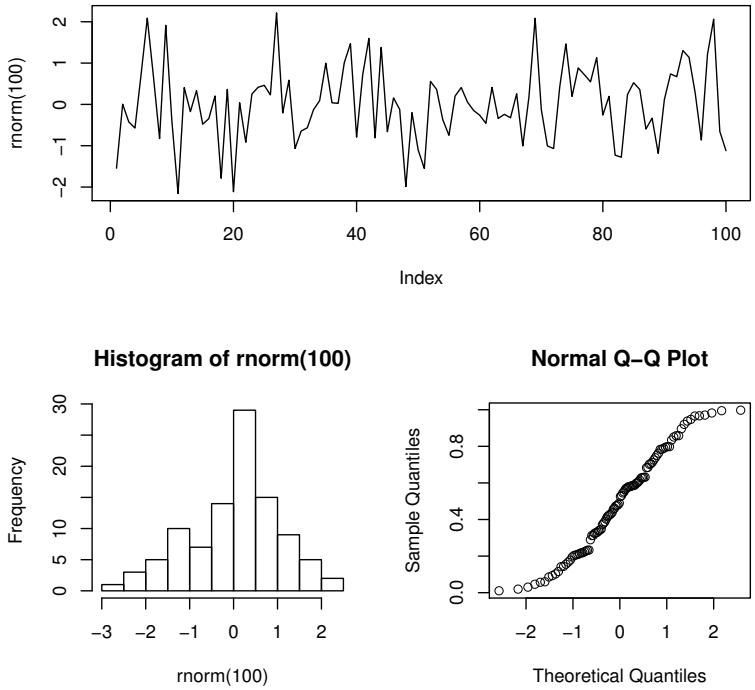

FIGURE 11.3: Plot layout. The plotting area of this graph was divided with the `layout()` function.

```
> nf = layout(rbind(c(1, 1), c(2, 3)))
```

Note that if you are not sure how layout has divided the window use the function `layout.show()` to display the window splits

```
> plot(rnorm(100), type = "l")
> hist(rnorm(100))
> qqnorm(runif(100))
```

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)))
```

11.4 MORE ABOUT COLOURS

The R environment has in its default package several function to handle colours.

Named colours

The function `colors()` returns the built-in color names which R knows about. These 657 color names can be used with a `col=` specification in graphics functions such as `plot()`. The list too long to be printed, but let us show the different shades of "blue"

```
> Colors = colors()
> Colors[grep("blue", Colors)]
 [1] "aliceblue"       "blue"            "blue1"           "blue2"
 [5] "blue3"           "blue4"           "blueviolet"      "cadetblue"
 [9] "cadetblue1"      "cadetblue2"      "cadetblue3"      "cadetblue4"
[13] "cornflowerblue"  "darkblue"        "darkslateblue"   "deepskyblue"
[17] "deepskyblue1"    "deepskyblue2"    "deepskyblue3"    "deepskyblue4"
[21] "dodgerblue"      "dodgerblue1"     "dodgerblue2"     "dodgerblue3"
[25] "dodgerblue4"     "lightblue"       "lightblue1"      "lightblue2"
[29] "lightblue3"      "lightblue4"      "lightskyblue"    "lightskyblue1"
[33] "lightskyblue2"   "lightskyblue3"   "lightskyblue4"   "lightslateblue"
[37] "lightsteelblue"  "lightsteelblue1" "lightsteelblue2" "lightsteelblue3"
[41] "lightsteelblue4" "mediumblue"      "mediumslateblue" "midnightblue"
[45] "navyblue"        "powderblue"      "royalblue"       "royalblue1"
[49] "royalblue2"      "royalblue3"      "royalblue4"      "skyblue"
[53] "skyblue1"        "skyblue2"        "skyblue3"        "skyblue4"
[57] "slateblue"       "slateblue1"      "slateblue2"      "slateblue3"
[61] "slateblue4"      "steelblue"       "steelblue1"      "steelblue2"
[65] "steelblue3"      "steelblue4"
```

An even wider variety of colours can be used from derived colour palettes such as `rainbow()`, `heat.colors()`, etc. , or can be created with primitives `rgb()` and `hsv()`.

Colour palettes

With the function `palette()` you can view or manipulate the color palette which is used when the function argument `col=` has a numeric index. Without any argument the function returns the current colour palette in use

```
> palette()
[1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"
[8] "gray"
```

The argument

```
> args(palette)
```

```
function (value)
NULL
```

If the argument `value` has length 1, it is taken to be the name of a built in colour palette. If `value` has length greater than 1 it is assumed to contain a description of the colors which are to make up the new palette, either by name or by red, green, blue, RGB, levels.

LISTING 11.8: A LIST OF R'S COLOUR PALETTES

```
Function:
rainbow             Rainbow colour palette
heat.colors         Heat colours palette
terrain.colors      Terrain colours palette
topo.colors         Topographic colours palette
cm.colors           CM colours palette
```

The following call creates 25 rainbow colours

```
> myRainbow = rainbow(25)
> palette(myRainbow)
> matplot(outer(1:100, 1:25), type = "l", lty = 1, lwd = 2, col = 1:25,
     main = "Rainbow Colors")
```

Here the R function `matplot()` plots the columns of one matrix against the columns of another.
The next example shows how to create a gray scale of 25 gray levels

```
> myGrays = gray(seq(0, 0.9, length = 25))
> palette(myGrays)
> matplot(outer(1:100, 1:25), type = "l", lty = 1, lwd = 2, col = 1:25,
     main = "Grays")
```

To reset to the default palette type

```
> palette("default")
```

Conceptually, all of the aolor palette functions use a line cut or part of a line cut out of the 3-dimensional color space. Some applications such as contouring require a palette of colors which do not wrap around to give a final color close to the starting one. With rainbow,

```
> args(rainbow)
function (n, s = 1, v = 1, start = 0, end = max(1, n - 1)/n,
    gamma = 1, alpha = 1)
NULL
```

the parameters start and end can be used to specify particular subranges. The following values can be used when generating such a subrange: red=0, yellow=1/6, green=2/6, cyan=3/6, blue=4/6 and magenta=5/6. The following plot shows some color wheels

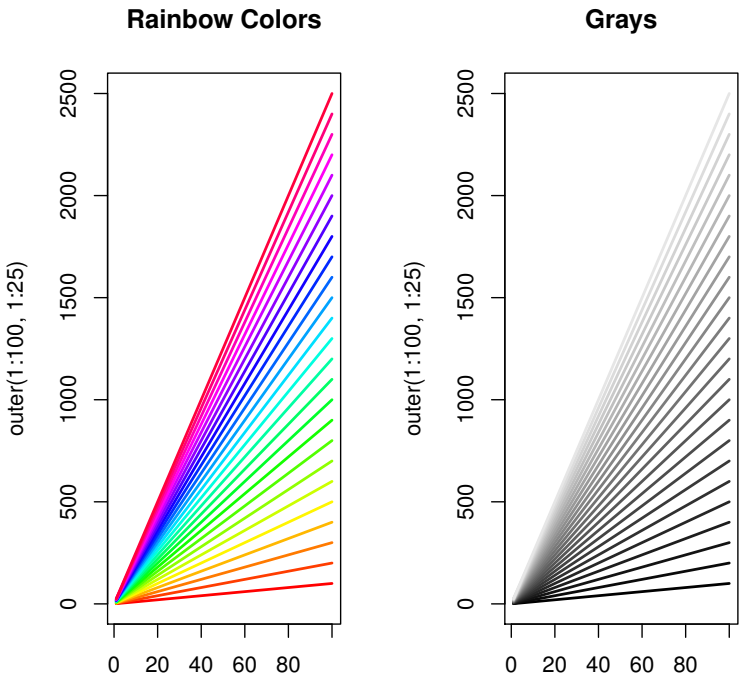

FIGURE 11.4: A rainbow color palette and a gray scale palette.

```
> par(mfrow = c(2, 2))
> pie(rep(1, 12), radius = 1, col = rainbow(12))
> pie(rep(1, 12), radius = 1, col = heat.colors(12))
> pie(rep(1, 12), radius = 1, col = topo.colors(12))
> pie(rep(1, 12), radius = 1, col = gray(seq(0, 0.9, length = 12)))
```

11.5 ADDING GRAPHICAL ELEMENTS TO AN EXISTING PLOT

Once you have created a plot you may want to add something to it. This can be done with low-level plot functions.

LISTING 11.9: A LIST OF PLOTTING FUNCTIONS TO ADD GRAPHICAL ELEMENTS TO AN EXISTING PLOT

```
Function:
points          adds points to a plot
lines           adds connected line segments to a plot
abline          adds straight lines through a plot
```

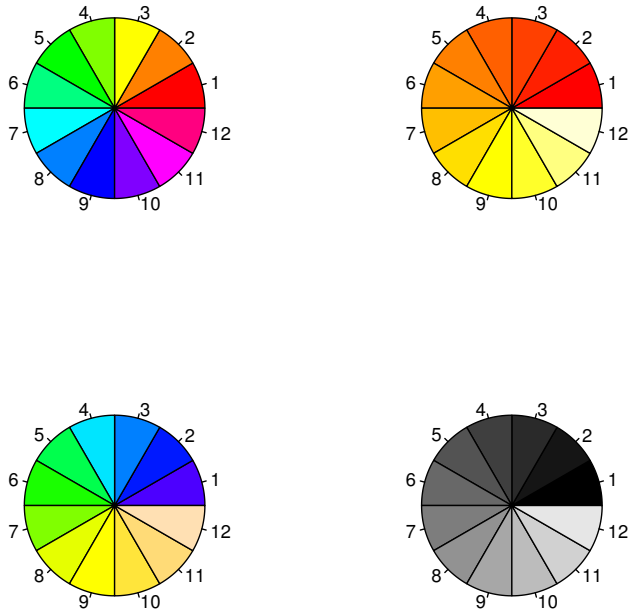

FIGURE 11.5: Colour Wheels from the rainbow, heat, topo, and gray scale palettes.

`arrows`	adds arrows between pairs of points
`title`	adds a title to a plot
`text`	adds text to a plot at the specified coordinates
`mtext`	adds text in the margins of a plot
`legend`	adds a legend to a plot

Adding lines

The functions `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)
```

Adding arrows and line segments

The functions `arrows()` and `segments()` are used to draw arrows and line segments.

```
> 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

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

The function `title` can be used to add a title, a subtitle and/or x- and y-labels to an already existing plot.

```
> title(main = "My title", sub = "My Subtitle")
```

Adding text

The function `text()` can be used to add text to an existing plot.

```
> text(0, 0, "Some Text")
> text(1, 1, "Rotated Text", 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.

Adding margin text

The function `mtext` is used to place text in one of the four margins of the plot.

```
> mtext("Text in the margin", side = 4, col = "grey")
```

Adding a legend

The function `legend()` ...

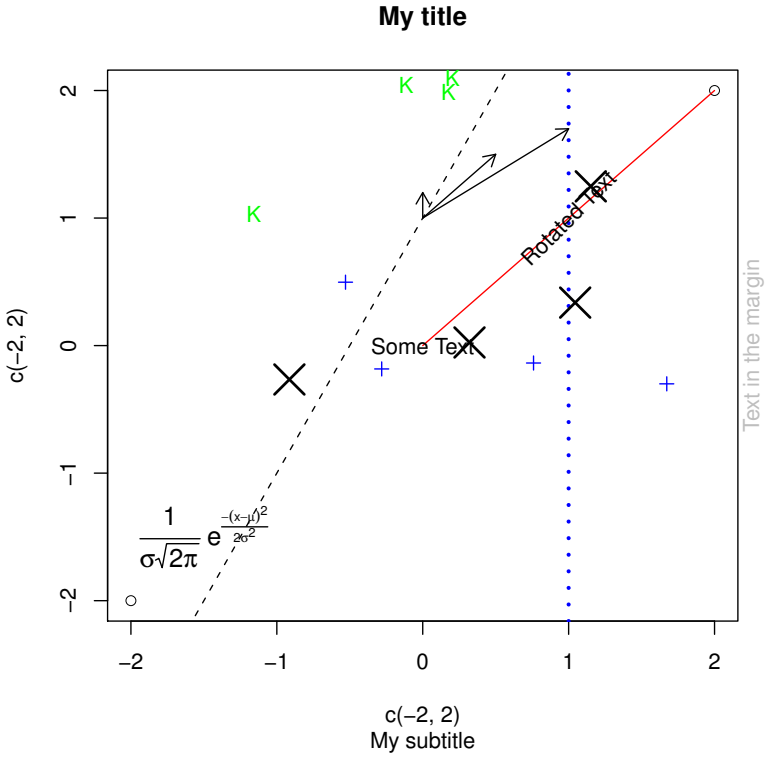

FIGURE 11.6: The graph that results from the previous low-level plot functions.

Adding mathematical expressions in graphs

In R you can place ordinary text on plots, but also special symbols, Greek characters and mathematical formulae 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.

11.6 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.

The `axis` argument

One approach would be to first create the plot without the axis with the `axes = FALSE` argument, and then draw the axis using the low-level `axis` function.

```
> x <- rnorm(100)
> y <- rnorm(100)
> plot(x, y, axes = FALSE)
> axis(side = 1)
> axis(side = 2)
```

The arguments `side` and `pos`

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 = FALSE)
> axis(side = 1, pos = 0)
> axis(side = 2, pos = 0)
```

The arguments `at` and `labels`

The location of the tick marks and the labels at the tick marks can be specified with the arguments `at` and `labels` respectively.

```
> x <- rnorm(100)
> y <- rnorm(100)
> plot(x, y, axes = FALSE)
> xtickplaces <- seq(-2, 2, l = 8)
> ytickplaces <- seq(-2, 2, l = 6)
> axis(side = 1, at = xtickplaces)
> axis(side = 2, at = ytickplaces)
```

```
> x <- 1:20
> y <- rnorm(20)
> plot(x, y, axes = FALSE)
> 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 = FALSE)
> 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)
```

The argument `tck`

Another useful parameter that you can use is the `tck` argument. 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.

Logarithmic axis style

To draw logarithmic x or y axis use `log = "x"` or `log = "y"`, if both axis need to be logarithmic use `log = "xy"`.

```
> axis(side = 1, at = c(5, 10, 15, 20), labels = rep("", 4), tck = 1,
      lty = 2)
> x <- runif(100, 1, 1e+05)
> y <- runif(100, 1, 1e+05)
> plot(x, y, log = "xy", col = "grey")
```

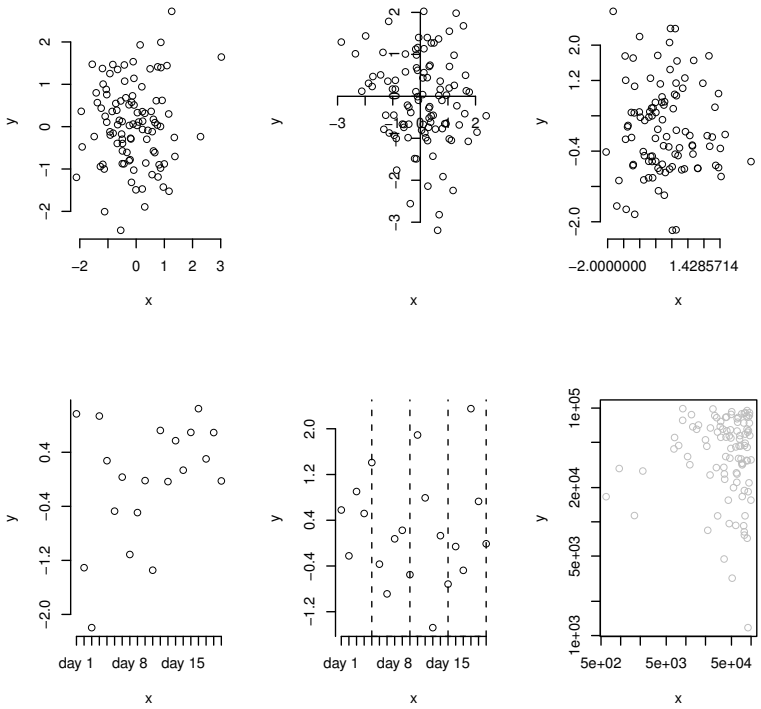

FIGURE 11.7: Examples of axis controls

CHAPTER 12

GRAPHICAL DEVICES

Before a graph can be generated 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, for example. Type

```
> ?Devices
```

for an overview of all available devices.

12.1 AVAILABLE DEVICES

The devices in R are:

LISTING 12.1: R'S GRAPHICS DEVICES.

```
Device:
windows             The graphics driver for Windows
postscript          Writes PostScript graphics to a file
pdf                 Writes PDF graphics to a file
pictex              Writes LaTeX/PicTeX graphics to a file
png                 The PNG bitmap device
jpeg                The JPEG bitmap device
bmp                 The BMP bitmap device
xfig                Device for XFIG graphics file format
bitmap              bitmap pseudo-device via GhostScript
x11                 The graphics device for the X11 Window system
quartz              The native graphics device on Mac OS X
cairo_pdf           PDF device based on cairo graphics
cairo_ps            PS device based on cairo graphics
```

12.2 DEVICE MANAGEMENT UNDER WINDOWS

How to show the default device

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.

How to open a new device

We could, however, also open a device ourselves first. The advantages of this are:

1. we can open the device without using the default values
2. 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.

Let us open two devices the first for a scatter plot and the second for a histogram plot.

```
> dev.new()
> plot(rnorm(100))

> dev.new()
> hist(rnorm(100))
```

Now two devices are open. In the first we have the scatter plot and in the second we have the histogram plot.

How to show a list of open devices

The number of open devices can be obtained by using the function `dev.list()`

```
> dev.list()
```

Which is the current active device

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()
```

Low-level plot commands are placed on the active device. In the above example the command

```
> title("Scatter Plot")
```

will result in a title on the active graph.

How to make another device active

Another device can be made active by using the function `dev.set()`.

```
> dev.set(which = 2)
> title("Histogram Plot")
```

How to close devices

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 png file so that it can be used in a web site, use the png device:

```
> png("test.png")
> plot(rnorm(100))
> dev.off()
```

12.3 LIST OF DEVICE FUNCTIONS

Here is a summary list of R's device functions

LISTING 12.2: R'S DEVICE FUNCTIONS.

```
Function:
dev.cur             returns number and name of the active device
dev.list            returns the numbers of all open devices
dev.next            returns number and name of the next device
dev.prev            returns number and name of the previous device
dev.off             shuts down the specified device
dev.set             makes the specified device the active device
dev.new             opens a new device
graphics.off        shuts down all open graphics devices
```

PART IV

STATISTICS AND INFERENCE

CHAPTER 13

BASIC STATISTICAL FUNCTIONS

Financial returns are generated from a stochastic process. To compare the returns from different financial instruments we can investigate them in several directions. What are the basic statistical properties, can we find a distribution function which fits the financial returns properly, what are the fitted parameters, are the returns drawn from a normal distribution? We can ask such and many other questions to compute and test the statistical properties of financial returns.
The base installation of R contains many functions for calculating statistical summaries, for data analysis and for statistical modelling. Even more functions are available in all the contributed R packages on CRAN and in the Rmetrics packages.
In this section we will present and discuss some of these functions und will explore to some extent the possibilities we have to analyse real world financial data sets.

13.1 STATISTICAL SUMMARIES

A number of functions return statistical summaries. The following table contains a list of only some of the statistical functions provided by R. The names of the functions usually speak for themselves.

LISTING 13.1: SOME FUNCTIONS THAT CALCULATE STATISTICAL SUMMARIES

```
Function:
acf                 auto or partial correlation coefficients
cor                 correlation coefficient
mad                 median absolute deviation
mean                mean and trimmed mean
median              median
quantile            sample quantile at given probabilities
range               the range, i.e. the vector  c(min(x), max(x))
var, cov            Variance and Covariance functions
```

Example: Hedge funds location and dispersion measures

As an example we compute location and dispersion measures from monthly performance measures of hedge funds in the year 2005. The data are taken from the web *www.hennesseegroup.com* where one can find also the data for the years after 2005 and for additional indices. The monthly return values of four stock matrket indices are added for comparison. The presented data set can be downloaded from the Rmetrics web site.

```
> data.frame(HedgeFund, row.names = NULL)
```

```
   JAN05 FEB05 MAR05 APR05 MAY05 JUN05 JUL05 AUG05 SEP05 OCT05 NOV05 DEC05
1  -0.91  1.26 -1.18 -1.95  1.88  1.87  2.65  0.39  1.33 -1.65  1.64  1.37
2  -0.10  0.90 -0.31 -1.55  0.05  1.22  1.75  0.89  1.10 -0.82  0.82  1.20
3   0.59  2.28 -1.07 -0.73  1.00  1.33  1.87  1.21  3.48 -1.72  2.23  3.05
4  -0.91 -0.48 -1.37 -2.89 -1.42  1.07  1.21  0.73  1.23 -0.21 -0.09  0.75
5   0.33  1.36  0.34 -0.60 -0.09  1.14  1.70  1.16  1.43 -0.42  1.20  1.62
6  -0.18  1.69 -0.40 -2.15  0.83  1.80  2.42  1.18  1.37 -2.62  1.42  1.62
7   0.30  1.48 -0.85 -0.99  0.46  1.33  1.73  1.16  0.65 -0.29  0.75  1.15
8   0.55  0.73  0.03 -0.20  0.43  0.70  0.44  0.26  0.91 -0.16  0.65  0.31
9   0.13  0.91  0.44 -1.04  0.75  0.86  1.24  0.57  0.01 -1.51  1.25  1.03
10  0.06  0.96 -0.17 -1.38 -0.29  0.92  2.01  0.97  0.96 -0.58  0.56  1.38
11 -0.89  1.65 -0.74 -2.04  0.77  2.13  2.01  1.10  2.57 -1.94  1.14  1.79
12  4.33  2.01  1.93  3.59 -3.97 -0.57 -1.53  2.25  2.35  2.69 -2.84 -0.20
13 -1.88  4.14 -2.89 -2.73 -0.38  1.12  3.02  2.26  4.27 -2.97  2.25  4.61
14 -5.20 -0.52 -2.56 -3.88  7.63 -0.54  6.22 -1.50 -0.02 -1.46  5.31 -1.23
15 -4.17  1.69 -2.86 -5.73  6.55  3.86  6.34 -1.85  0.31 -3.10  4.85 -0.46
16 -2.53  1.89 -1.91 -2.01  3.00 -0.01  3.60 -1.12  0.70 -1.77  3.52 -0.10
```

For printing reasons, here comes the legend

```
> data.frame(Instrument = rownames(HedgeFund))
```

```
            Instrument
1       LongShortEquity
2  ArbitrageEventDriven
3           GlobalMacro
4  ConvertibleArbitrage
5            Distressed
6           EventDriven
7             HighYield
8         MarketNeutral
9       MergerArbitrage
10    MultipleArbitrage
11        Opportunistic
12          ShortBiased
13            MSCI.EAFE
14               NASDAQ
15          Russell2000
16                SP500
```

Now we want to calculate the basic statistics of the indices and to present them in a table. These include the mean, `mean()`, the standard deviation, `sd()`, and the minimum, `min()` and maximum, `max()`, values.

```
> Mean = round(apply(HedgeFund, 1, mean), 2)
> Sdev = round(apply(HedgeFund, 1, sd), 2)
> Min = apply(HedgeFund, 1, min)
> Max = apply(HedgeFund, 1, max)
```

Now bind the statistics to a data frame.

```
> Statistics <- data.frame(cbind(Mean, Sdev, Min, Max))
> Statistics
                      Mean Sdev   Min  Max
LongShortEquity       0.56 1.57 -1.95 2.65
ArbitrageEventDriven  0.43 0.97 -1.55 1.75
GlobalMacro           1.13 1.63 -1.72 3.48
ConvertibleArbitrage -0.20 1.28 -2.89 1.23
Distressed            0.76 0.81 -0.60 1.70
EventDriven           0.58 1.60 -2.62 2.42
HighYield             0.57 0.89 -0.99 1.73
MarketNeutral         0.39 0.35 -0.20 0.91
MergerArbitrage       0.39 0.87 -1.51 1.25
MultipleArbitrage     0.45 0.94 -1.38 2.01
Opportunistic         0.63 1.62 -2.04 2.57
ShortBiased           0.84 2.62 -3.97 4.33
MSCI.EAFE             0.90 2.95 -2.97 4.61
NASDAQ                0.19 4.04 -5.20 7.63
Russell2000           0.45 4.20 -5.73 6.55
SP500                 0.27 2.26 -2.53 3.60
```

It is worth noting that the function `summary()` can also be convenient for calculating basic statistics of columns of a data frame.
In the base `R` package we also find robust measures like the trimmed mean, the `median()` or `mad()`, and Huber's estimators in the MASS package.

13.2 DISTRIBUTION FUNCTIONS

Most of the common probability distributions are implemented in `R`'s base package. Each distribution has implemented four functions:

1. the cumulative probability distribution function
2. the probability density function
3. the quantile function
4. a random sample generator

Function naming conventions

The names of these functions consist of the code for the distribution preceded by a letter indicating the desired task

LISTING 13.2: LETTERS PRECEDING DISTRIBUTION FUNCTIONS AND THEIR MEANING

```
Letter:
p                   for the probability distribution function
d                   for the density function
q                   for the quantile function
r                   for the random number generator
```

For example, the corresponding commands for the normal distribution are:

```
dnorm(x, mean = 0, sd = 1)
pnorm(q, mean = 0, sd = 1)
qnorm(p ,mean = 0, sd = 1)
rnorm(n, mean = 0, sd = 1)
```

In these expressions `mean` and `sd` 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.

Distributions in R's base environment

The next table gives an overview of the available distribution functions in R. Don't forget to precede the code with `d`, `p`, `q` or `r` (for example `pbeta()` or `qgamma()`).

LISTING 13.3: PROBABILITY DISTRIBUTION FUNCTIONS IN R

```
Function:
beta                Beta distribution
binom               Binomial distribution
cauchy              Cauchy distribution
chisq               chi squared distribution
exp                 exponential distribution
f                   F distribution
gamma               Gamma distribution
geom                Geometric distribution
hyper               Hyper geometric distribution
lnorm               Lognormal distribution
logis               Logistic distribution
nbinom              Negative binomial distribution
norm                Normal (Gaussian) distribution
pois                Poisson distribution
```

```
t                   Student's t distribution
unif                Uniform distribution
weibull             Weibull distribution
wilcoxon            Wilcoxon distribution
```

Example: The distribution of NYSE composite Index returns

In this example we want to express the NYSE Composite Index returns by a normal distribution function and compare the curve with the empirical histogram

```
> NYSE = diff(log(nyse[, 2]))
> hist(NYSE, probability = TRUE, breaks = "FD", col = "steelblue",
     xlim = c(-0.05, 0.05))
> x = seq(-0.05, 0.05, length = 251)
> lines(x, dnorm(x, mean(NYSE), sd(NYSE)), col = "orange")
```

We observe essential differences between the empirical and fitted data. The empirical data ar much more peaked and have heavier tails. This is a stylized fact known as leptokurtic behavior of financial returns.

13.3 RANDOM NUMBERS

Random number generation

The following code generates 1000 random numbers from the standard normal distribution with 5% contamination, using the `ifelse()` function.

```
> x <- rnorm(n = 1000)
> cont <- rnorm(n = 1000, mean = 0, sd = 10)
> p <- runif(n = 1000)
> z <- ifelse(p < 0.95, x, cont)
```

Sampling random numbers

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:

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

To randomly sample three elements from `x` use

```
> sample(x, 3)
```

```
[1] 45  4 38
```

To sample three elements from x with replacement use

```
> sample(x, 3, rep = TRUE)
[1] 23  5 24
```

Random number seeds

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] -1.48057  1.57717 -0.95674 -0.92001 -1.99764
> rnorm(5)
[1] -0.27230 -0.31535 -0.62826 -0.10646  0.42801
> set.seed(12)
> rnorm(5)
[1] -1.48057  1.57717 -0.95674 -0.92001 -1.99764
```

13.4 HYPOTHESIS TESTING

Hypothesis testing is a method of making statistical decisions from empirical data. Statistically significant results are unlikely to occur by chance. R comes with several statistical functions for hypothesis testing. These include

LISTING 13.4: SOME FUNCTIONS FOR HYPOTHESIS TESTING

```
Function:
ks.test             Kolmogorov-Smirnov goodness of fit test
t.test              One or two sample Student's t-test
chisq.test          chi squared goodness of fit test
var.test            test on variance equality of x and y
```

Distribution tests

To test if a data vector is drawn from a certain distribution the function `ks.test()` can be used. In the function

```
> args(ks.test)
function (x, y, ..., alternative = c("two.sided", "less", "greater"),
    exact = NULL)
NULL
```

the first argument `x` describesd a numeric vector of data values. The second argument `y` holds either a numeric vector of data values, or a character string naming a cumulative distribution function or an actual cumulative distribution function such as `"pnorm"`. If `y` is numeric, a two-sample test of the null hypothesis that `x` and `y` were drawn from the same continuous distribution is performed. Alternatively, when `y` is a character string naming a continuous cumulative distribution function, then a one-sample test is carried out of the null that the distribution function which generated `x` is distribution `y` with parameters specified by the dots argument. Now let us show an example for this case

```
> x <- runif(100)
> test = ks.test(x, "pnorm")
> test

One-sample Kolmogorov-Smirnov test

data:  x
D = 0.5183, p-value < 2.2e-16
alternative hypothesis: two-sided
```

Compare with

```
> x <- rnorm(100)
> test = ks.test(x, "pnorm")
> test

One-sample Kolmogorov-Smirnov test

data:  x
D = 0.0613, p-value = 0.8469
alternative hypothesis: two-sided
```

In the first excample the the very small p-value says us that the hypothesis that `x` is drawn from a normal distribution is rejected, and in the second example the hypothesis is accepted.
The output object `out` is an object of class 'htest'. It is a list with five components.

```
> names(test)
[1] "statistic"   "p.value"     "alternative" "method"      "data.name"

> test$statistic
```

```
       D
0.061279
```

Now let us use the function 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 = 0.08, p-value = 0.9062
alternative hypothesis: two-sided
```

and

```
> x1 = rnorm(100)
> x2 = runif(100)
> ks.test(x1, x2)

Two-sample Kolmogorov-Smirnov test

data:  x1 and x2
D = 0.5, p-value = 2.778e-11
alternative hypothesis: two-sided
```

Alternative functions that can be used are `chisq.test()`,`shapiro.test()` and `wilcox.test()`.

13.5 PARAMETER ESTIMATION

How to fit the parameters of a distribution

The recommende package `MASS` has a function to fit the parameters of a distribution function. `MASS` is part of the base environment of `R`.

```
> require(MASS)
```

If we assume that random variates are drawn from a Weibull distribution with given shpae and scale parameters, we proceed in the follow way to estimate these parameters

```
> x <- rweibull(100, shape = 4, scale = 100)
> fitdistr(x, "weibull")

     shape       scale
    4.20913   100.44812
 (  0.32201) (  2.51407)
```

13.6 DISTRIBUTION TAILS AND 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 $\mathcal{N}(0,1)$ distribution, proceed as follows:

```
> x <- rnorm(100)
> q <- quantile(x, c(0.05, 0.1))
> q
      5%     10%
-1.4982 -1.2373
```

The function returns a vector with the quantiles as named elements.

CHAPTER 14

LINEAR TIME SERIES ANALYSIS

R's base and stats packages have a broad spectrum of functions implemented which are useful in time series modelling, forecasting as well as for a diagnostic analysis of the fitted models. We will select some of these functions and show how to use them for fitting the parameters of linear time series models.

14.1 OVERVIEW OF FUNCTIONS FOR TIME SERIES ANALYSIS

The following listing gives a selective overview of functions from R's `base` and stats `package` which are useful for time series analysis.

LISTING 14.1: R FUNCTIONS FOR ARMA TIME SERIES ANALYSIS

```
Function:
ar                  fit an AR model selecting the complexity by AIC
 ar.ols              fit an AR model by the ordinary least square approach
 filter              can be used to simulate from an AR model
 predict             predict from a model fitted by the ar function
arima               fit an ARIMA model to a univariate time series
 arima.sim           simulate from an ARIMA model
 predict             forecast from models fitted by arima
tsdiag              shows diagnostic plots from the fitted time series
acf                 computes and plots the autocorrelation function
pacf                the same for the partial autocorrelation function
Box.test            computes Box-Pierce and Ljung-Box hypothestis tests
```

This group of functions includes functions with several methods for the parameter estimation of autoregressive models, `ar()`, as well as for their extension with moving average terms, `arima()` setting the `order=c(m,0,n)`, and for the case of integrated models, `arima()`. The functions `ar()` and `arima()` return the fitted model parameters, called *coefficients*, and the estimated value of its *sigma squared* a measure for the quality of the

fitted model. In the case of the `arima()` function also the value of the log-likelihood estimator, and the AIC criterion are printed.
There are als functions for forecasting, `predict()`, from an estimated model. `predict()` is a generic function for predictions from the results of various model fitting functions. The function invokes particular methods which depend on the class of the first argument. For example the `ar()` returns an object of class "ar", and the function `arima()` of class "Arima". A call to the function `predict()` thus executes the generic functions `predict.ar()` and `predict.Arima()`, respectively.
For a diagnostic analys of the fitted models we can perform hypothesis tests on the fitted residuals and them also in diagnostic plots. The diagnostic plot created by the function `tsdiag()` shows three graphs the standardized residuals, the atocorrelationfunction, ACF, of the residuals, and the p-Values for the Ljung-Box Statistics. The results returned by the function `tsdiag()` were created from the functions `acf()` and `Box.test()`.

14.2 SIMULATION FROM AN AUTOREGRESSIVE PRORCESS

The coefficients for an autoregressive process are defined through the equation

$$x_t - \mu = a_1(x_{t-1} - \mu) + \cdots + a_p(x_{t-p} - \mu) + \varepsilon_t$$

Here, x_t are the observable data points, a_k are the coefficients of the generating process, and ε_t are innovations from an external noise term which we assum to be normal distributed.
The function `filter()`

```
> args(filter)
function (x, filter, method = c("convolution", "recursive"),
    sides = 2, circular = FALSE, init = NULL)
NULL
```

applies linear filtering, for booth autoregressiv, AR, and moving average, MA, processes, to a univariate time series or to each series separately of a multivariate time series. Here we will use this function generate an AR time series process.

LISTING 14.2: ARGUMENTS OF THE FUNCTION FILTER(). FOR A FULL AND DETAILED DESCRIPTION WE REFER TO THE AR HELP PAGE.

```
Argument:
x           aunivariate or multivariate time series
filter      vector of filter coefficients in reverse time order
method      "convolution" applies MA, "recursive" AR filtering
sides       decides on the use of 1 or 2 sided convolution filters
circular    decides to wrap a convolution filter around the ends
init        specifies the intial values for a recursive filter
```

In the help page we find the defitions for the convolution and recursive filters. Note that there is an implied coefficient 1 at lag 0 in the recursive filter, which gives

$$y[i] = x[i] + f[1] * y[i-1] + ... + f[p] * y[i-p]$$

No check is made from the choice of your parameters to see if recursive filter is invertible or not: the output may diverge if it is not.
The convolution filter is defined as

$$y[i] = f[1] * x[i+o] + ... + f[p] * x[i+o-(p-1)]$$

where o is the offset which is determined from the *sides* arguments.
If we want to simulate from an AR(2) process we have to set the argument `method="recursive"`, the coefficients for example to `filter=c(-0.5,0.25)`, and the starting values to `init=c(0,0)`.

```
> set.seed(4711)
> eps = rnorm(100)
> filter(x = eps, filter = c(-0.5, 0.25), method = "r", init = c(0,
    0))
Time Series:
Start = 1
End = 100
Frequency = 1
  [1]  1.819735  0.460572  1.420966 -1.002219  0.245372 -1.882153  1.819969
  [8] -2.345191  1.583065 -0.903476 -0.134662 -1.730648  0.866990 -0.443008
 [15]  0.640480  1.145215 -0.881530  1.106649 -1.308333  1.962546 -0.598789
 [22]  0.868480  0.392143 -0.241038 -1.151741  0.711856 -1.856219  0.947218
 [29] -1.605967  1.922391 -1.144393  2.818342 -0.552850  0.389813 -0.950950
 [36] -0.188089 -0.431225 -2.390784 -0.752176  1.450996 -2.186195  1.457216
 [43] -1.506742 -0.830052  0.640795  0.214630  0.287064  1.859636 -0.038196
 [50] -1.236918  0.270421  0.094018 -0.447051 -1.250823 -0.022581 -1.209217
 [57]  0.281930 -2.444528  2.019774 -0.680794  1.354014 -0.054551  0.508934
 [64] -0.290913  0.222811 -1.111663  0.008365 -1.055445  1.491222 -1.011139
 [71]  0.650895 -0.384876  1.118933 -0.489480  0.606742  0.679393  0.223238
 [78]  2.451229  0.756272  0.224620 -0.153271 -0.223091  0.631278  0.714437
 [85]  0.277104 -2.006861  1.396770 -0.944298  1.364612 -2.018208 -0.214554
 [92] -0.596543  0.724706 -0.331283  1.125702 -1.506480  0.500603 -1.048720
 [99]  2.192371 -1.973837
```

We have fixed the seed, so that you will get the same innovations as those used in this course. For comparison let us compute the series step by step recursively

```
> f = c(-0.5, 0.25)
> x = eps
> x[1] = eps[1]
> x[2] = eps[2] + f[2] * x[2 - 1]
> for (i in 3:100) {
      for (k in 1:2) {
          x[i] <- x[i] + f[k] * x[i - k]
```

```
    }
 }
> ar.ts = ts(x)
> ar.ts
Time Series:
Start = 1
End = 100
Frequency = 1
  [1]  1.819735  1.825373  0.738565 -0.319819 -0.266428 -1.455653  1.478769
  [8] -2.067965  1.359153 -0.722213 -0.281272 -1.612028  0.771027 -0.365372
 [15]  0.577672  1.196029 -0.922639  1.139907 -1.335239  1.984313 -0.616400
 [22]  0.882727  0.380617 -0.231714 -1.159285  0.717959 -1.861157  0.951212
 [29] -1.609199  1.925006 -1.146508  2.820053 -0.554235  0.390933 -0.951856
 [36] -0.187356 -0.431818 -2.390304 -0.752564  1.451310 -2.186449  1.457422
 [43] -1.506908 -0.829917  0.640686  0.214718  0.286993  1.859693 -0.038243
 [50] -1.236881  0.270390  0.094043 -0.447071 -1.250806 -0.022594 -1.209206
 [57]  0.281922 -2.444521  2.019769 -0.680790  1.354010 -0.054548  0.508932
 [64] -0.290911  0.222810 -1.111662  0.008364 -1.055444  1.491221 -1.011138
 [71]  0.650894 -0.384876  1.118933 -0.489479  0.606742  0.679393  0.223238
 [78]  2.451229  0.756272  0.224620 -0.153271 -0.223091  0.631278  0.714437
 [85]  0.277104 -2.006861  1.396770 -0.944298  1.364612 -2.018208 -0.214554
 [92] -0.596543  0.724706 -0.331283  1.125702 -1.506480  0.500603 -1.048720
 [99]  2.192371 -1.973837
```

After having defined the filter coefficients and the innovations eps_t we looped in time (3:100) and order (1:2) over the series to create the observations. the last line converts the numeric vector x into a time series object of class `ts`. Let us plot a longer series with 500 points and its cumulated series.

```
> plot(ar.ts, type = "l")
> abline(h = 0, col = "grey", lty = 3)


> ari.ts = ts(cumsum(ar.ts))
> plot(ari.ts, type = "l")
> grid(col = "grey")
```

The function `acf()` measures the cross-correlation of a time series with itself. Therfore the function gots the name *autocorrelation function* or short ACF(k) for a given lag k. The ACF searches for repeating patterns in a time series. In our AR(3) example we should observe that heigboured values are negatively correlated and leading to an oscillating decay of the ACF.

```
> acf(ar.ts, xlim = c(0, 10))
```

If the coefficient in our AR(2) example would not have been negative, we would not have been expect the oscillating decay.

```
> f <- c(0.5, 0.25)
> ar2.ts <- filter(x = eps, filter = f, method = "r", init = c(0,
     0))
> acf(ar2.ts, xlim = c(0, 10))
```

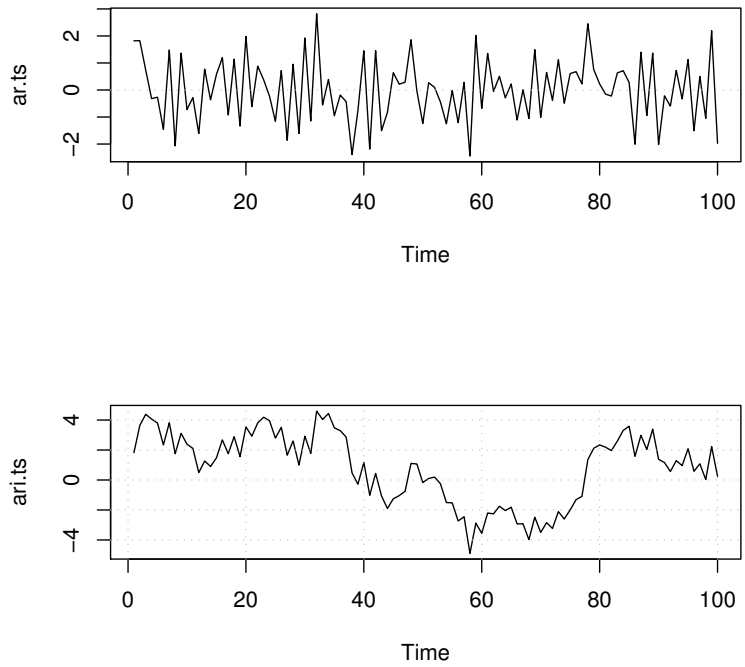

FIGURE 14.1: Simulated AR(2) time series plot.

The partial autocorrelation function `pacf()` of lag k, PACF(k), is the ACF between x_t and x_{t+k} with the linear dependence of x_{t+1} through to x_{t+k-1} removed. For the first moment this construction sounds unmotivated, but if we consider an AR(p) process in detail then lags greater than then the order p vanish. Thus PACFs are useful in identifying the order of an AR model.

```
> pacf(ar.ts, xlim = c(1, 10))
> abline(v = 2.5, col = "red", lty = 3)


> pacf(ar2.ts, xlim = c(1, 10))
> abline(v = 2.5, col = "red", lty = 3)
```

The figure confirms a maximum order of 2 for the the simulated AR(2) models.

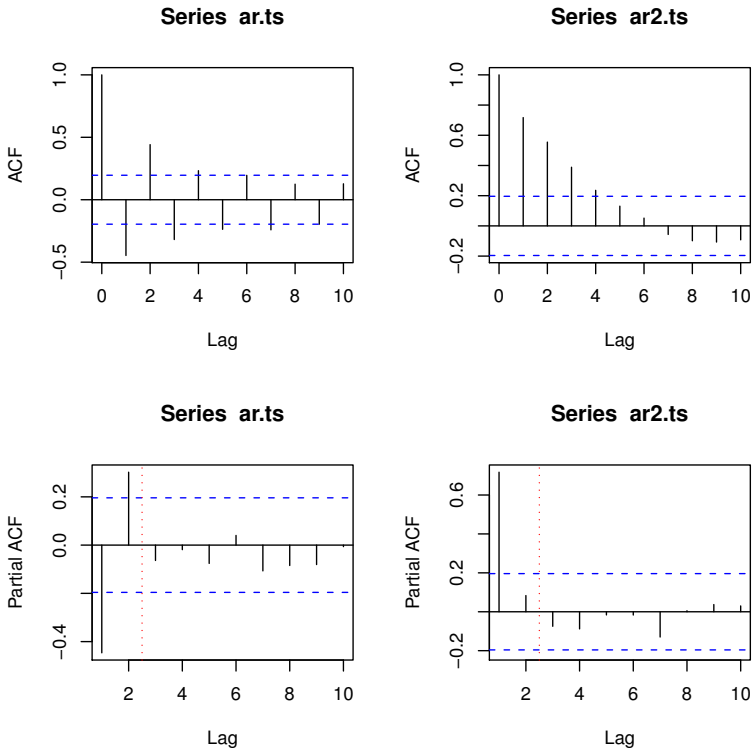

FIGURE 14.2: ACF and PACF plots for an AR(2) model

14.3 AR - FITTING AUTOREGRESSIVE MODELS

The function `ar()` fits an autoregressive time series model to empirical data by ordinary least squares, by default selecting the complexity by the Akaike Information Criterion Statistics, AIC.
AIC, beside other ICs, measures the goodness of the fit of an estimated statistical model. The criterion is based on the concept of entropy offering a relative measure of the *information* lost when a given model is used to describe reality. In this sense the measure describes the tradeoff between the bias and variance in model construction looking for models with low complexitiy but high precision. A model with a lower AIC has a higher preference compared to a model with a larger AIC value.
The estimation of the AR model parameters is done by the R function `ar()`

```
> args(ar)
function (x, aic = TRUE, order.max = NULL, method = c("yule-walker",
    "burg", "ols", "mle", "yw"), na.action = na.fail, series = deparse(substitute(x)),
    ...)
NULL
```

LISTING 14.3: ARGUMENTS OF THE FUNCTION AR(). FOR A FULL AND DETAILED DESCRIPTION WE REFER TO THE AR HELP PAGE

```
Argument:
x           a univariate or multivariate time series
aic         ff TRUE then AIC otherwise order.max is fitted
order.max   maximum order of model to fit
method      giving the method used to fit the model
na.action   function to be called to handle missing values
series      names for the series
...         additional arguments for specific methods
demean      should a mean be estimated during fitting,
            passed to the underlying methods
```

Several methods for the parameter estimation are provided by R. Here we concentrate on the methods `method="mle"`, which performs a *maximum log-likelihood estimation*, and the `method="ols"` which does an *ordinary least square estimation*.
For both variants order selection for the parameter `p` is done by AIC if `aic=TRUE`. This is problematic for the OLS method since the AIC is computed as if the variance estimate were the MLE.
The provided implementation of `ar()` includes by default a constant in the model, by removing the overall mean of `x` before fitting the AR model, or estimating (MLE) a constant to subtract.

Example U.S. GNP data

As an example let us fit the parameters of the quarterly growth rates of the U.S. real gross national product, GNP, seasonally adjusted. The data can be obtained from the FRED2 data base:

```
> name = "GNP"
> URL = paste("http://research.stlouisfed.org/fred2/series/", name,
     "/", "downloaddata/", name, ".csv", sep = "")
> download = read.csv(URL)
> head(download)
        DATE VALUE
1 1947-01-01 238.1
2 1947-04-01 241.5
3 1947-07-01 245.6
4 1947-10-01 255.6
5 1948-01-01 261.7
6 1948-04-01 268.7
```

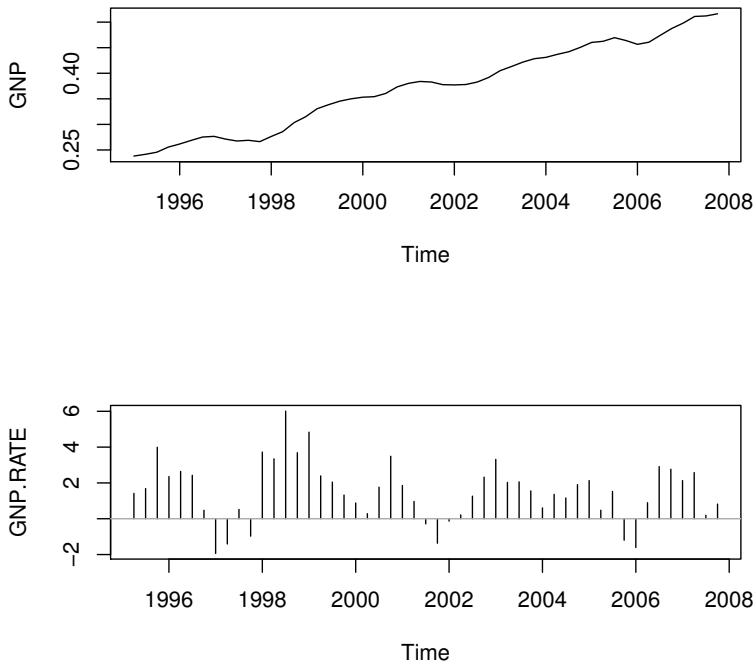

FIGURE 14.3: Quarterly GNP time series plot.

We transform the data from the download into a time series object. Taking only data points before 2008 we exclude the data points from the recent subprime crisis. We also measure the GNP in units of 1000.

```
> GNP = ts(download[1:52, 2]/1000, start = c(1995, 1), freq = 4)
> GNP.RATE = 100 * diff(log(GNP))
```

Before we start to model the GNP, let us have a look at the GNP time series and its growth rate.

```
> plot(GNP, type = "l")
> plot(GNP.RATE, type = "h")
> abline(h = 0, col = "darkgray")
```

AR parameter estimation: We get the following parameter estimates for the fitted parameters from the ordinary least square, OLS, estimator

```
> gnpFit = ar(GNP.RATE, method = "ols")
> gnpFit
Call:
ar(x = GNP.RATE, method = "ols")

Coefficients:
     1       2       3       4       5       6       7       8       9      10
 0.442  -0.021  -0.159  -0.124  -0.128   0.186  -0.090  -0.058   0.215  -0.194
    11      12      13      14      15      16      17
-0.121  -0.008   0.014   0.093  -0.231   0.076   0.160

Intercept: -0.214 (0.145)

Order selected 17  sigma^2 estimated as  0.65
```

The result of the estimation tells us that the best fit was achieved by an AR(2) model.
There is more information available as provided by the printing.

```
> class(gnpFit)
[1] "ar"
> names(gnpFit)
 [1] "order"       "ar"          "var.pred"    "x.mean"      "x.intercept"
 [6] "aic"         "n.used"      "order.max"   "partialacf"  "resid"
[11] "method"      "series"      "frequency"   "call"        "asy.se.coef"
```

The returned object of the function `ar()` is a list with the following entries

LISTING 14.4: VALUES OF THE FUNCTION AR(). FOR A FULL AND DETAILED DESCRIPTION WE REFER TO THE AR HELP PAGE.

```
Values:
order           order of the fitted model chosen by the AIC
ar              estimated autoregression coefficients
var.pred        prediction variance
x.mean          estimated mean of the series used in fitting
x.intercept     model intercept in x-x.mean, for OLS only
aic             value of the aic argument.
n.used          number of observations in the time series
order.max       value of the order.max argument
partialacf      estimate of the pacf up to lag order.max
resid           residuals from the fitted model
method          name of the selected method
series          name(s) of the time series
frequency       frequency of the time series
call            the matched call.
asy.var.coef    asymptotic variance of estimated coefficients
```

Order selection: The best order selection by AIC suggested $p = 2$. This can be confirmed by a view of the partial autocorrelation function, which dies out after two lags. Have a look at the plot.

```
> pacf(GNP.RATE)
```

14.4 AUTOREGRESSIVE MOVING AVERAGE MODELLING

Autoregressive moving average modelling, ARMA, adds a moving average part, MA, to the pure autoregressive process, therefore the name ARMA. Allowing to difference (integrate) the models, we get the so called integrate ARMA, or short ARIMA, models.
In the literature different definitions of ARMA models have different signs for the AR and/or MA coefficients. The definition used by R is[1]

$$x_t = a_1 x_{t-1} + \cdots + a_p x_{t-p} + \varepsilon_t + b_1 \varepsilon_{t-1} + \cdots + b_q \varepsilon_{t-q}$$

The function `arima()` from R's stats package allows you to analyze these kinds of linear time series models.

```
> args(arima)
function (x, order = c(0, 0, 0), seasonal = list(order = c(0,
    0, 0), period = NA), xreg = NULL, include.mean = TRUE, transform.pars = TRUE,
    fixed = NULL, init = NULL, method = c("CSS-ML", "ML", "CSS"),
    n.cond, optim.method = "BFGS", optim.control = list(), kappa = 1e+06)
NULL
```

LISTING 14.5: ARGUMENTS OF THE FUNCTION ARIMA. FOR A FULL AND DETAILED DESCRIPTION WE REFER TO THE ARIMA HELP PAGE.

```
Argument:
x                   a univariate time series
order               order of the non-seasonal part of the ARIMA model
seasonal            specifies the seasonal part of the ARIMA model
xreg                a vector or matrix of optional external regressors
include.mean        should the ARMA model include a mean/intercept term
transform.pars      transforms AR parameters to ensure stationarity
fixed               allows to fix coefficients
init                optional numeric vector of initial parameter values
method              determines the fitting method
n.cond              for CSS method number of initial observations to ignore
optim.control       List of control parameters for function optim()
kappa               prior variance for past observations in differenced
     models
```

ARIMA parameter estimation

Fitting the GNP growth rate with an ARMA(2,1) model, we get

[1] and therefore the MA coefficients differ in sign from those of S-PLUS.

```
> gnpFit = arima(GNP.RATE, order = c(2, 0, 1))
> gnpFit
Call:
arima(x = GNP.RATE, order = c(2, 0, 1))

Coefficients:
         ar1    ar2    ma1  intercept
      -0.144  0.477  0.662      1.484
s.e.   0.271  0.167  0.269      0.449

sigma^2 estimated as 1.75:  log likelihood = -86.93,  aic = 183.85
```

Diagnostic analysis

To get a better idea on the quality of the fitted model parameters we can investigate the residuals of the fitted model using the function `tsdiag()`.

```
> tsdiag(gnpFit)
```

LISTING 14.6: ARGUMENTS OF THE FUNCTION TSDIAG. FOR A FULL AND DETAILED DESCRIPTION WE REFER TO THE TSDIAG HELP PAGE.

```
Argument:
object              a fitted time-series model
gof.lag             maximum lags for a Portmanteau goodness-of-fit test
...                 further arguments to be passed to particular methods
```

The residuals from a fitted model can be extracted using the generic function `residuals()`. The abbreviated form `resid()` is an alias for `residuals()`.

```
> args(residuals)
function (object, ...)
NULL
```

LISTING 14.7: ARGUMENTS OF THE FUNCTION RESIDUALS. FOR A FULL AND DETAILED DESCRIPTION WE REFER TO THE TSDIAG HELP PAGE.

```
Argument:
object              a fitted object from which to extraction model
     residuals
...                 other arguments to be passed
```

```
> gnpResid <- residuals(gnpFit)
> gnpResid
          Qtr1      Qtr2      Qtr3      Qtr4
1995           -0.054242  0.234237  2.402407
1996 -0.442305  0.377841  0.441628 -1.720508
```

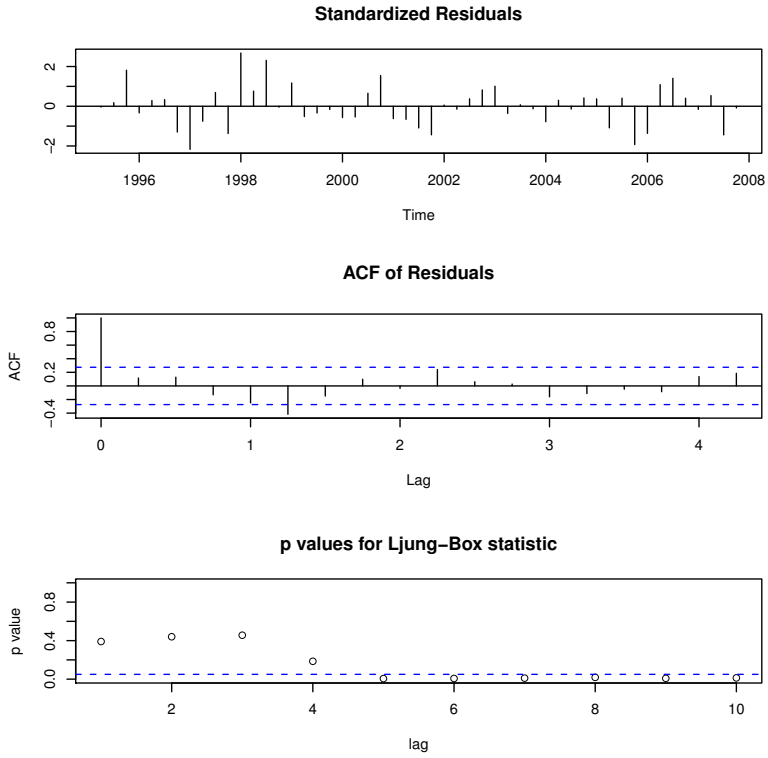

FIGURE 14.4: Quarterly GNP time series diagnostics.

```
1997 -2.875129 -1.000601  0.913589 -1.819013
1998  3.547771  1.004371  3.060105 -0.058494
1999  1.549204 -0.686327 -0.450260 -0.215312
2000 -0.751778 -0.713571  0.865501  2.044735
2001 -0.824612 -0.872334 -1.446071 -1.903232
2002  0.077242 -0.196654  0.495404  1.086237
2003  1.336968 -0.477844  0.097598 -0.173489
2004 -1.029039  0.397257 -0.185773  0.555974
2005  0.492367 -1.440910  0.540241 -2.555669
2006 -1.805101  1.439335  1.861324  0.535498
2007 -0.203375  0.708847 -1.909115 -0.109896
```

```
> par(mfrow = c(2, 2))
> hist(gnpResid, probability = TRUE, breaks = "FD", xlim = c(-1.5,
      1.5), col = "steelblue", border = "white")
> x = seq(-2, 2, length = 100)
```

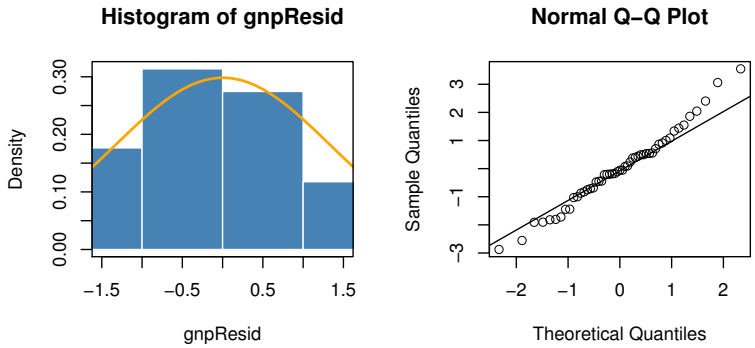

FIGURE 14.5: Quarterly GNP residuals plots from a fitted AR(2) model.

```
> lines(x, dnorm(x, mean = mean(gnpResid), sd = sd(gnpResid)),
      col = "orange", lwd = 2)
> box()
> qqnorm(gnpResid)
> qqline(gnpResid)
```

14.5 FORECASTING FROM ESTIMATED MODELS

Finally we will close this short investigation by forecasting the growth rates 1 year or four quarters ahead

```
> predict(gnpFit, n.ahead = 4)
$pred
        Qtr1    Qtr2    Qtr3    Qtr4
2008 0.89314 1.25120 1.23603 1.40896
```

```
$se
       Qtr1   Qtr2   Qtr3   Qtr4
2008 1.3246 1.4918 1.5841 1.6038
```

CHAPTER 15

REGRESSION MODELING

In statistics, regression analysis refers to techniques for modelling and analyzing several variables, when the focus is on the relationship between a dependent variable (response) and one or more independent variables. Regression analysis can thus be used for prediction including forecasting of time-series data. Furthermore, regression analysis is also used to understand which among the independent variables are related to the dependent variable, and to explore the forms of these relationships. A large body of techniques for carrying out regression analysis has been developed. Familiar methods such as linear regression and ordinary least squares regression are parametric, in that the regression function is defined in terms of a finite number of unknown parameters that are estimated from the data. In this and the following chapters we will show how to use basic regression modeling for financial applications.

15.1 LINEAR REGRESSION MODELS

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 modelled as a normal distribution with some variance.

The function `lm()` and its arguments

In base R the function `lm()` is used to fit linear models and to carry out regression.

```
> args(lm)
```

```
function (formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)
NULL
```

The function comes with quite a few arguments which have the following meaning:

- formula an object of class "formula" (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under 'Details'.
- data an optional data frame, list or environment (or object coercible by as.data.frame to a data frame) containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which lm is called.
- subset an optional vector specifying a subset of observations to be used in the fitting process.
- weights an optional vector of weights to be used in the fitting process. Should be NULL or a numeric vector. If non-NULL, weighted least squares is used with weights (that is, minimizing `sum(`$w * e^2$`)`); otherwise ordinary least squares is used.
- na.action a function which indicates what should happen when the data contain NAs. The default is set by the na.action setting of options, and is na.fail if that is unset. The default is na.omit. Another possible value is NULL, no action. Value na.exclude can be useful.
- method the method to be used; for fitting, currently only the method = "qr" is supported; method = "model.frame" returns the model frame (the same as with model = TRUE, see below).
- model, x, y, qr logicals. If TRUE the corresponding components of the fit (the model frame, the model matrix, the response, the QR decomposition) are returned.
- singular.ok logical. If FALSE (the default in S but not in R) a singular fit is an error.
- contrasts an optional list. See the contrasts.arg of model.matrix.default.
- offset this can be used to specify an a priori known component to be included in the linear predictor during fitting. This should be NULL or a numeric vector of length either one or equal to the number of cases. One or more offset terms can be included in the formula instead or as well, and if both are specified their sum is used. See model.offset.

- ... additional arguments to be passed to the low level regression fitting functions (see below).

The first argument needs some more explanation.

Formula objects - How to formulate a regression model

Formula objects play a very important role in statistical modeling in R, they are used to specify a model to be fitted. The exact meaning of a formula object depends on the modeling function. We will look at some examples in the context of linear modelling. The general form for a formula is given by:

```
response ~ expression
```

Sometimes the term `response` can be omitted. The term `expression` is a collection of variables combined by *operators*. Two typical examples of formula objects are

```
> formula <- y ~ x1 + x2
> formula
y ~ x1 + x2
> class(formula)
[1] "formula"
```

and a somehow more complex formula

```
> formula2 <- log(y) ~ log(x1) + log(x2) + x2:x3
> formula2
log(y) ~ log(x1) + log(x2) + x2:x3
```

A description of formulating models using formulae is given in the help page of `formula()`. By default R includes the intercept of the linear regression model. To omit the intercept use the formula:

```
> y ~ -1 + x1 + x2
y ~ -1 + x1 + x2
```

R offers many additional operators to express formula relationships. Be aware of the special meaning of such operators `*`, `-`, `^`, `\` and `:` in linear model formulae. Important, they are not used for the normal multiplication, subtraction, power and division.
For example the `:` operator is used to model interaction terms in linear models. The next formula example includes an interaction term between the variable x_1 and the variable x_2

```
> y ~ x1 + x2 + x1:x2
```

```
y ~ x1 + x2 + x1:x2
```

which corresponds to the linear regression model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + \epsilon$$

Note that there is a short hand notation for the above formula which is given by

```
> y ~ x1 * x2
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.

```
> formula <- y ~ x1 * x2
> terms(formula)
y ~ x1 * x2
attr(,"variables")
list(y, x1, x2)
attr(,"factors")
   x1 x2 x1:x2
y   0  0     0
x1  1  0     1
x2  0  1     1
attr(,"term.labels")
[1] "x1"    "x2"    "x1:x2"
attr(,"order")
[1] 1 1 2
attr(,"intercept")
[1] 1
attr(,"response")
[1] 1
attr(,".Environment")
<environment: R_GlobalEnv>
```

The ^ operator is used to generate interaction terms up to a certain order. For example the formula expression

```
> y ~ x1 + x2 + +x3 + x1:x2 + x2:x3 + x1:x3
y ~ x1 + x2 + +x3 + x1:x2 + x2:x3 + x1:x3
```

is also equivalent to

```
> y ~ (x1 + x2 + x3)^2
y ~ (x1 + x2 + x3)^2
```

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
y ~ (x1 + x2 + x3)^2 - x2:x3
```

which is equivalent to

```
> y ~ x1 + x2 + x3 + x1:x2 + x1:x3
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
y ~ x1 + 2 * x2
```

The $*$ operator already has a specific meaning, so you should use the following construction:

```
> y ~ x1 + I(2 * x2)
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)
y ~ x1 + (x2 - constant)
```

Use the following formula instead:

```
> y ~ x1 + I(x2 - constant)
y ~ x1 + I(x2 - constant)
```

15.2 PARAMETER ESTIMATION

Now we are ready to formulate linear regression models. Let us start with a very simple model the Capital Asset Pricing Model, CAPM, in portfolio optimization.

The Capital Asset Pricing Model

The CAPM model says that the return to investors has to be equal to the risk-free rate plus a premium for stocks as a whole that is higher than the risk-free rate. This can be expressed as a linear model

$$R_c = r_f + \beta(r_M - r_f)$$

where, R_c is the company's expected return on capital, r_f is the risk-free return rate, usually a long-term U.S. Treasury bill rate, and r_M is the expected return on the entire market of all investments. β is called the company's beta.
Most measures for β use a common broad stock market index over the past 5 or 10 years.
What is Beta?: Relying on the assumption that markets are efficient, we see investors vote every day on whether the stock market will rise or fall. Watching a long series of measures we should be able to see a relationship between a change in stock prices and the market.

Example: Betas for the Dow Jones equities

In this example we want to compute the betas for the equities of the Dow Jones Index. As the market reference we use the SP500 Index, and for the risk free rate the 3 months T-Bills.

Loading the DJ example file

The data set comes with the example data files in the R package for this book.

```
> data(DowJones)
> DJ = as.timeSeries(DowJones)
> dim(DJ)
[1] 2529   30
> range(DJ)
[1]   1.18 137.04
> names(DJ)
 [1] "AA"   "AXP"  "T"    "BA"   "CAT"  "C"    "KO"   "DD"   "EK"   "XOM"
[11] "GE"   "GM"   "HWP"  "HD"   "HON"  "INTC" "IBM"  "IP"   "JPM"  "JNJ"
[21] "MCD"  "MRK"  "MSFT" "MMM"  "MO"   "PG"   "SBC"  "UTX"  "WMT"  "DIS"

> DJ30 = alignDailySeries(DJ)
> dim(DJ30)
[1] 2612   30
```

Downloading from Yahoo the market index

In the next step we want to download from Yahoo Finance the market index of the SP500 as a representant of a broad market index . But first we write a small function named `getYahoo()` to download the series from the Internet and to transform it into an object of class `timeSeries`.

```
> getYahoo <- function(Symbol, start = c(y = 1990, m = 1, d = 1)) {
    stopifnot(length(Symbol) == 1)
    URL <- paste("http://chart.yahoo.com/table.csv?", "s=", Symbol,
        "&a=", start[2] - 1, "&b=", start[3], "&c=", start[1],
        "&d=", 11, "&e=", 31, "&f=", 2099, "&g=d&q=q&y=0", "&z=",
        Symbol, "&x=.csv", sep = "")
```

```
    x = read.csv(URL)
    X = timeSeries(data = data.matrix(x[, 2:7]), charvec = as.character(x[,
        1]), units = paste(Symbol, names(x)[-1], sep = "."))
    rev(X)
}
```

Now we are ready for the download. After the download we cut the SP500 series to the same length as series of the DJ30 equities, select the "Open" column and rename it to "SP500"

```
> SP500 = getYahoo("^GSPC")
> SP500 = window(SP500, start(DJ), end(DJ30))
> SP500 = SP500[, "^GSPC.Close"]
> names(SP500) <- "SP500"
> head(SP500, 10)
GMT
            SP500
1990-12-31 330.22
1991-01-02 326.45
1991-01-03 321.91
1991-01-04 321.00
1991-01-07 315.44
1991-01-08 314.90
1991-01-09 311.49
1991-01-10 314.53
1991-01-11 315.23
1991-01-14 312.49
> range(time(SP500))
GMT
[1] [1990-12-31] [2001-01-02]
> nrow(SP500)
[1] 2529
```

Finally we align the SP500 to weekdays.

```
> SP500 = alignDailySeries(SP500)
> dim(SP500)
[1] 2612    1
```

Downloading from Fred the risk free market rate

We proceed in the same way as we downloaded the SP500. The T-Bill series can be downloaded from the Fed's data base server FRED in St. Louis. The symbol name is "DTB3" and the URL is composed in the function `getFRed()`.

```
> getFred <- function(Symbol) {
    stopifnot(length(Symbol) == 1)
    URL <- paste("http://research.stlouisfed.org/fred2/series/",
        Symbol, "/downloaddata/", Symbol, ".csv", sep = "")
    x = read.csv(URL)
    X = timeSeries(data = as.numeric(data.matrix(x[, 2])), charvec = as.character(x[,
```

```
        1]), units = Symbol)
    na.omit(X)
}
```

Download the data and prepare them to be merged with the SP500 and DJ30 time series.

```
> DTB3 = getFred("DTB3")
> DTB3 = window(DTB3, start(DJ), end(DJ30))
> head(DTB3, 10)
```

```
GMT
           DTB3
1990-12-31 6.44
1991-01-02 6.46
1991-01-03 6.44
1991-01-04 6.53
1991-01-07 6.51
1991-01-08 6.44
1991-01-09 6.25
1991-01-10 6.21
1991-01-11 6.16
1991-01-14 6.05
```

```
> DTB3 = alignDailySeries(DTB3)
> dim(DTB3)
```

```
[1] 2612    1
```

Compute the Betas

Compose the series of SP500, DTB3, and DJ30, take care of the proper annualized returns ...

...

15.3 MODEL DIAGNOSTICS

The object `recession.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:

```
> cars2.lm <- lm(Weight ~ Mileage, data = cars2)
> cars2.residuals <- resid(cars2.lm)
> ks.test(cars2.residuals, "pnorm", mean = mean(cars2.residuals),
     sd = sd(cars2.residuals))
```

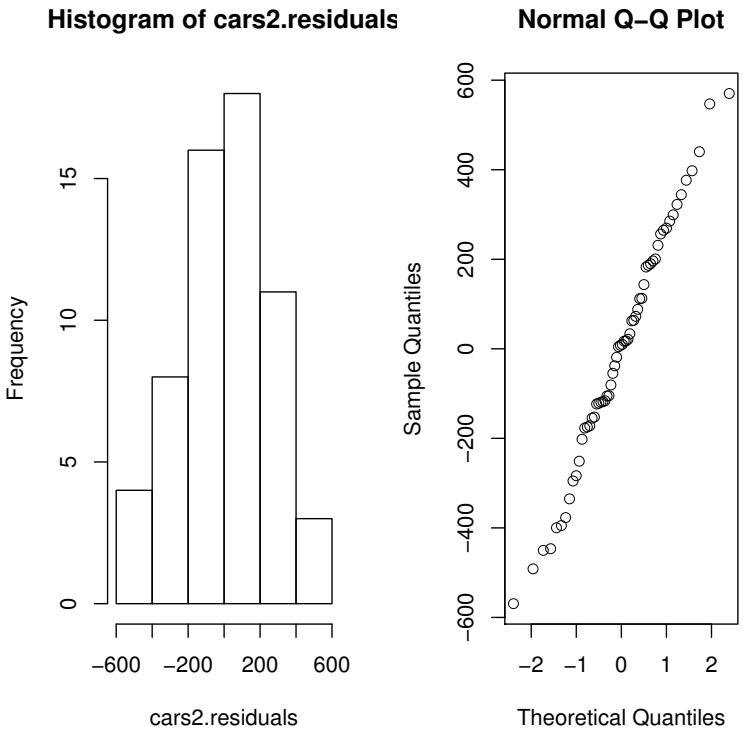

FIGURE 15.1: A histogram and a qq-plot of the model residuals to check normality of the residuals.

```
One-sample Kolmogorov-Smirnov test

data:  cars2.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(cars2.residuals)
> qqnorm(cars2.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.

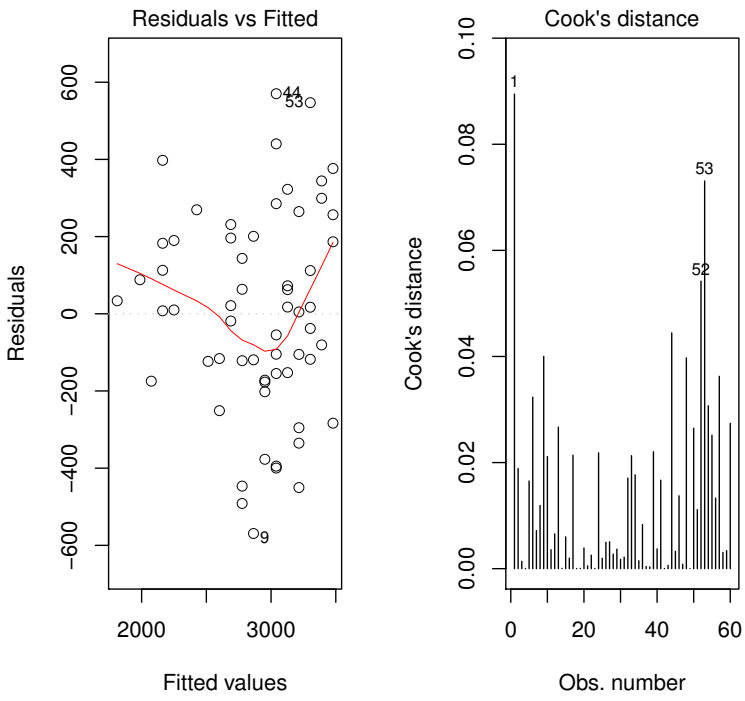

FIGURE 15.2: Diagnostic plots to check for linearity and for outliers.

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(cars2.lm, which = 1)
> plot(cars2.lm, which = 4)
```

15.4 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 a `Disp.` variable representing the engine displacement.

```
> add1(cars2.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(cars2.lm, ~Mileage)
Single term deletions

Model:
Weight ~ Mileage
        Df Sum of Sq      RSS AIC
<none>                4078578 672
Mileage  1  10428530 14507108 746
```

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(cars2.lm, ~. + Disp.)
> cars.lm2
Call:
lm(formula = Weight ~ Mileage + Disp., data = cars2)

Coefficients:
(Intercept)      Mileage        Disp.
     3748.4        -58.0          3.8
```

Forecasting U.S. recession from the yield curve

US Recession Data: The Chicago Fed National Activity Index (CFNAI) is a monthly index designed to better gauge overall economic activity and inflationary pressure. The CFNAI is released at the end of each calendar month.

The CFNAI is a weighted average of 85 existing monthly indicators of national economic activity. It is constructed to have an average value of zero and a standard deviation of one. Since economic activity tends toward trend growth rate over time, a positive index reading corresponds to growth above trend and a negative index reading corresponds to growth below trend.

Downloading the CFNAI from the Internet: There are two possibilities to download the data from the Chicago FED Internet site: (i) to download the data as an XLS file, to convert it in a CSV File and to store it on your computer. (ii) to download the HTML file, clean it from the tags and extract the data records. As an example we follow her the second option.

```
> URL = "http://www.chicagofed.org/webpages/research/data/cfnai/current_data.cfm"
> ans = readLines(URL)
```

Extract the data records from the HTML file and clean the data file, i.e. substitute tags and remove non-numeric lines

```
> data <- ans[grep("<.td>$", ans)]
> data <- gsub("<td>", "", data)
> data <- gsub("<.td>", "", data)
> data <- gsub("N.A", "0", data)
> data <- gsub(":", ".", data)
> data <- data[-grep("[A-Z]", data)]
```

Then convert into a numeric matrix, keep the proper columns

```
> data <- rev(as.numeric(data))
> data <- matrix(data, byrow = TRUE, ncol = 7)
> Data = data[, 2:1]
> colnames(Data) <- c("CFNAI", "CFNAI-MA3")
> rownames(Data) <- data[, 7]
```

and finally transform the matrix into an object of class `ts`.

```
> CFNAI = ts(data = Data, start = c(1967, 3), frequency = 12)
> plot(CFNAI)
```

US Short and Long Term Interest Rates: In 1996 Arturo Estrella and Frederic Mishkin discuss the yield curve as a predictor of US recessions. They showed that the spread between the interest rates on the ten-year Treasury note and the three-month Treasury bill is a valuable forecasting tool and outperforms other financial and macroeconomic indicators in predicting recessions two to six quarters ahead.
First we download monthly data of the three-month Treasury bill and the ten-year Treasury note from the FRED 2 data base.

```
> tBill = "TB3MS"
> tNote = "GS10"
> tBillURL <- paste("http://research.stlouisfed.org/fred2/series/",
      tBill, "/", "downloaddata/", tBill, ".csv", sep = "")
> tNoteURL <- paste("http://research.stlouisfed.org/fred2/series/",
      tNote, "/", "downloaddata/", tNote, ".csv", sep = "")
> tBillSeries = read.csv(tBillURL)
> tNoteSeries = read.csv(tNoteURL)
> tBillSeries = tBillSeries[-(1:grep("1964-05", tBillSeries[, 1])),
      2]
> tNoteSeries = tNoteSeries[-(1:grep("1964-05", tNoteSeries[, 1])),
      2]
```

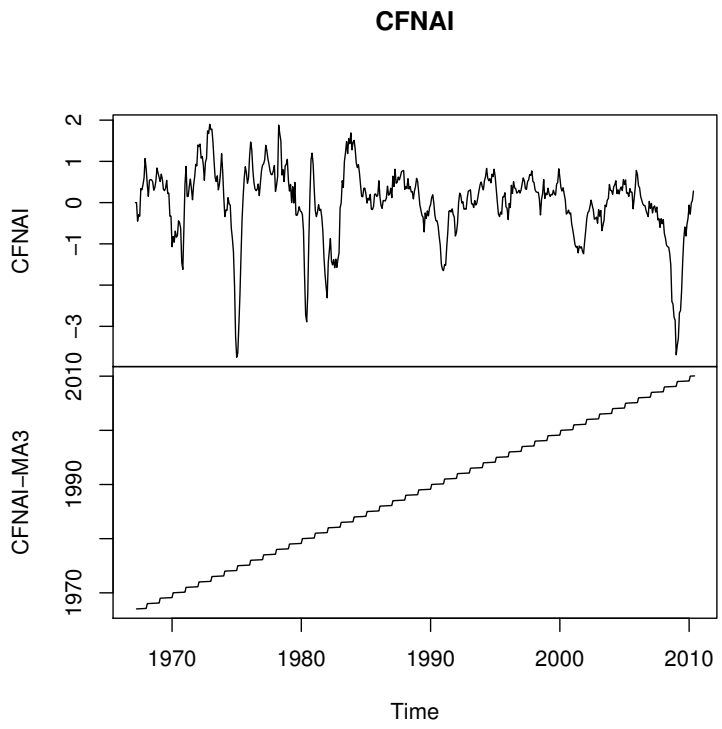

FIGURE 15.3: Plot of the Chicago Fed National Activity Index

```
> treasury = ts(cbind(tBillSeries, tNoteSeries, tSpread = tNoteSeries -
      tBillSeries), start = c(1964, 5), frequency = 12)
> plot(treasury)
```

Model the Data: recession short interest + long interests

As an example we will use our recession and interest rate data sets to fit the following linear regression model.

$$\texttt{Recession} = \beta_0 + \beta_1 \times \texttt{ShortTermInterests} + \beta_1 \times \texttt{LongTermInterests} \epsilon$$

In R this model is formulated and fitted as follows ...
So `recession.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.

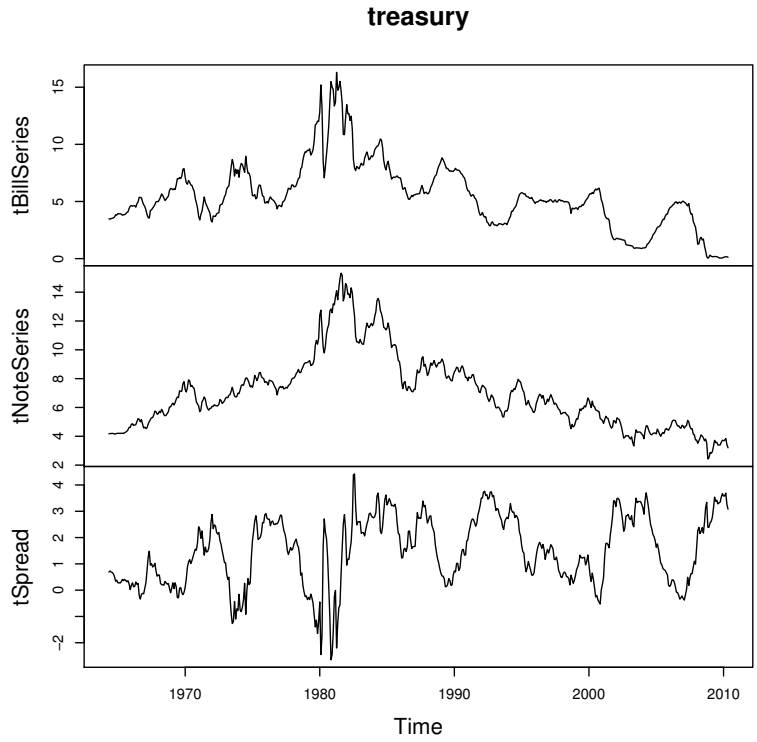

FIGURE 15.4: 3m tBills, 10y tNotes and tSpread

LISTING 15.1: LIST OF FUNCTIONS THAT ACCEPT AN LM OBJECT.

```
Generic function:
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)        creates diagnostic plots
```

These functions are generic. They will also work on objects returned by other statistical modelling 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.

CHAPTER 16

DISSIMILARITIES OF DATA RECORDS

For a multivariate data set many questions in finance ask which of the variables or data records are dissimilar. Can we cluster similar data records in groups or can we find an hierarchical or other kind of structure behind the observations made.
R's `graphics` and `stats` packages include several functions which help to answer these questions if data records are similar or to what extent they differ from each other. The `graphics` and `stats` packages include functions for computing correlations, for creating star and segment plots, and for clustering data into groups.

16.1 CORRELATIONS AND PAIRWISE PLOTS

In statistics, correlation indicates the strength and direction of a linear relationship between two random variables and measures their departure from independence. In this broad sense there are several coefficients, measuring the degree of correlation, adapted to the nature of the data.
In R the function `cor()` computes the correlation between two vectors `x` and `y` vectors. If `x` and `y` are matrices then the correlations between the columns of `x` and the columns of `y` are computed. In the same way the function `cov()` measures the covariances.
For an exploratory data analysis the function `pairs()` creates a matrix of scatterplots.

Example: World major stock market capitalizations

In the first example we ask the question: Which world major stock markets are similar? The demo data set named `Capitalization` contains for the years 2003 to 2008 the stock market capitalizations for 13 stock markets.

```
> Caps = Capitalization/1000
> rownames(Caps) = abbreviate(rownames(Caps), 6)
```

```
> Caps = ts(t(Caps), start = 2003, frequency = 1)
```

Note that we have expressed here the capitalizations as a time series object of class `ts` in units of 1000 USD and have abbreviated the corresponding stock market names for a more compact printing.

```
> Caps[, 1:7]
Time Series:
Start = 2003
End = 2008
Frequency = 1
      ErnxUS  TSXGrp  AstrSE  BmbySE  HngKSE  NSEInd  ShngSE
2003  1329.0  888.68  585.43  278.66  714.60  252.89  360.11
2004 12707.6 1177.52  776.40  386.32  861.46  363.28  314.31
2005  3632.3 1482.18  804.01  553.07 1055.00  515.97  286.19
2006 15421.2 1700.71 1095.86  818.88 1714.95  774.12  917.51
2007 15650.8 2186.55 1298.32 1819.10 2654.42 1660.10 3694.35
2008  9208.9 1033.45  683.87  647.20 1328.77  600.28 1425.35
```

and the remaining six series are

```
> Caps[, 8:13]
Time Series:
Start = 2003
End = 2008
Frequency = 1
     TokySE  BMESSE DtschB LndnSE ErnxEU  SIX SE
2003 2953.1  726.24 1079.0 2460.1 2076.4  727.10
2004 3557.7  940.67 1194.5 2865.2 2441.3  826.04
2005 4572.9  959.91 1221.1 3058.2 2706.8  935.45
2006 4614.1 1322.91 1637.6 3794.3 3712.7 1212.31
2007 4330.9 1781.13 2105.2 3851.7 4222.7 1271.05
2008 3115.8  948.35 1110.6 1868.2 2101.7  857.31
```

To group similar stock markets we first explore the pairwise correlations between the markets

```
> cor(Caps)
        ErnxUS  TSXGrp  AstrSE  BmbySE  HngKSE  NSEInd  ShngSE  TokySE  BMESSE
ErnxUS 1.00000 0.66304 0.78653 0.64182 0.71461 0.65205 0.57585 0.43570 0.77295
TSXGrp 0.66304 1.00000 0.97622 0.89808 0.90019 0.90341 0.74029 0.81474 0.95437
AstrSE 0.78653 0.97622 1.00000 0.89765 0.92404 0.90435 0.75838 0.74826 0.97869
BmbySE 0.64182 0.89808 0.89765 1.00000 0.98277 0.99981 0.96031 0.49666 0.96563
HngKSE 0.71461 0.90019 0.92404 0.98277 1.00000 0.98566 0.93245 0.53386 0.97779
NSEInd 0.65205 0.90341 0.90435 0.99981 0.98566 1.00000 0.95665 0.50840 0.96965
ShngSE 0.57585 0.74029 0.75838 0.96031 0.93245 0.95665 1.00000 0.23919 0.87535
TokySE 0.43570 0.81474 0.74826 0.49666 0.53386 0.50840 0.23919 1.00000 0.62992
BMESSE 0.77295 0.95437 0.97869 0.96563 0.97779 0.96965 0.87535 0.62992 1.00000
DtschB 0.72240 0.95461 0.97865 0.94422 0.95090 0.94743 0.84299 0.63443 0.98665
LndnSE 0.59734 0.88426 0.88688 0.64225 0.66316 0.65005 0.42743 0.84522 0.78587
ErnxEU 0.72434 0.97201 0.99052 0.86439 0.89426 0.87125 0.70899 0.77771 0.95382
SIX SE 0.75370 0.94971 0.97466 0.85312 0.91121 0.86243 0.70311 0.78990 0.94329
        DtschB  LndnSE  ErnxEU  SIX SE
ErnxUS 0.72240 0.59734 0.72434 0.75370
```

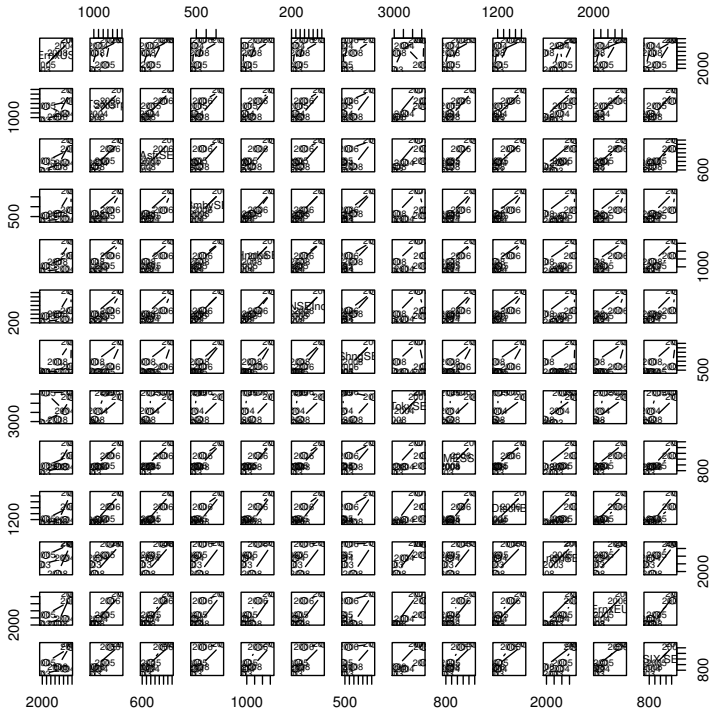

FIGURE 16.1: A pairs plot of correlations between the market capitalizations of the world's major stock markets in the years 2003 to 2008.

```
TSXGrp 0.95461 0.88426 0.97201 0.94971
AstrSE 0.97865 0.88688 0.99052 0.97466
BmbySE 0.94422 0.64225 0.86439 0.85312
HngKSE 0.95090 0.66316 0.89426 0.91121
NSEInd 0.94743 0.65005 0.87125 0.86243
ShngSE 0.84299 0.42743 0.70899 0.70311
TokySE 0.63443 0.84522 0.77771 0.78990
BMESSE 0.98665 0.78587 0.95382 0.94329
DtschB 1.00000 0.84321 0.97266 0.93594
LndnSE 0.84321 1.00000 0.92605 0.85672
ErnxEU 0.97266 0.92605 1.00000 0.97552
SIX SE 0.93594 0.85672 0.97552 1.00000

> pairs(Caps, pch = 19, cex = 0.8)
```

and then visualize them by a pairs plot.

It is not easy to compare pairwise correlations in the number matrix or even in the pairwise correlation plot when the number of elements becomes large. In the following we show an alternative image plot to display correlations from a large correlation matrix. The individual steps are the following:
Step 1: Convert the data into a matrix object, get the number of columns abbreviate the column and row names, and compute the correlations

```
> R = as.matrix(Caps)
> n = ncol(R)
> Names = abbreviate(colnames(R), 4)
> corr <- cor(R)
```

Step 2: Compute the appropriate colour for each pairwise correlation for the use in the image plot

```
> ncolors <- 10 * length(unique(as.vector(corr)))
> k <- round(ncolors/2)
> r <- c(rep(0, k), seq(0, 1, length = k))
> g <- c(rev(seq(0, 1, length = k)), rep(0, k))
> b <- rep(0, 2 * k)
> corrColorMatrix <- (rgb(r, g, b))
```

Step3: Plot the image, add axis labels, title and a box around the image

```
> image(x = 1:n, y = 1:n, z = corr[, n:1], col = corrColorMatrix,
     axes = FALSE, main = "", xlab = "", ylab = "")
> axis(2, at = n:1, labels = colnames(R), las = 2)
> axis(1, at = 1:n, labels = colnames(R), las = 2)
> title(main = "Pearson Correlation Image Matrix")
> box()
```

Step 4: Add the values of the pairwise correlations as text strings to each cell of the image plot

```
> x = y = 1:n
> nx = ny = length(y)
> xoy = cbind(rep(x, ny), as.vector(matrix(y, nx, ny, byrow = TRUE)))
> coord = matrix(xoy, nx * ny, 2, byrow = FALSE)
> X = t(corr)
> for (i in 1:(n * n)) {
     text(coord[i, 1], coord[n * n + 1 - i, 2], round(X[coord[i,
         1], coord[i, 2]], digits = 2), col = "white", cex = 0.7)
 }
```

It is left to the reader to build a function around the above code snippets.

Example: Pension fund benchmark portfolio

In a second example we explore the data set of assets classes which are part of the Swiss pension fund benchmark.
The Pictet LPP-Indices are a family of Benchmarks of Swiss Pension Funds. The LPP indices were created in 1985 by the Swiss Private Bank Pictet &

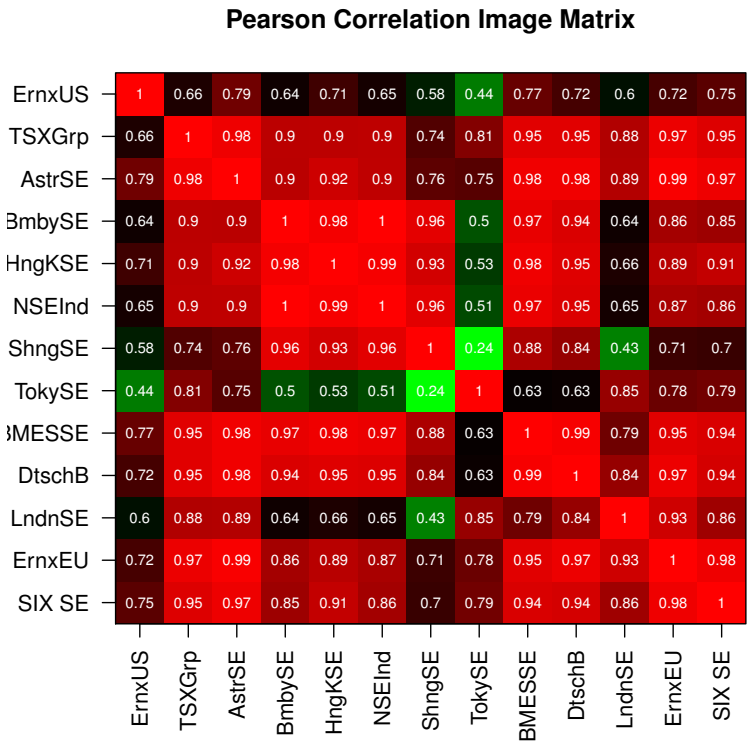

FIGURE 16.2: An image plot of correlations between the market capitalizations of the world's major stock markets in the years 2003 to 2008. The numbers in the squares are the computes values for the pairwise correlations, the colour visualizes the correlations: red shows high, and green shows low correlations.

Cie with the introduction of new Swiss regulations governing the investment of pension fund assets. Since then it has established itself as the authoritative pension fund index for Switzerland. Several adjustments have been made by launching new LPP-indices in 1993, 2000 and 2005. The LPP 2005 family of indices was introduced in 2005 and consists of three members named LPP25, LPP40 and LPP60 describing three indices with an increasing risk profile. The data set considered here covers the daily log-returns of three benchmark series with low, medium and high risk, and 6 asset classes from which they were constructed.

```
> data(PensionFund)
```

this returned a `data.frame` object, but we prefer to coerce the pension fund data into a `timeSeries` object

```
> PensionFund = 100 * as.timeSeries(PensionFund)
```

```
> head(round(PensionFund, 3), 10)
GMT
               SBI    SPI    SII    LMI    MPI    ALT  LPP25  LPP40  LPP60
2005-11-01 -0.061  0.841 -0.319 -0.111  0.155 -0.257 -0.013  0.020  0.081
2005-11-02 -0.276  0.252 -0.412 -0.118  0.034 -0.114 -0.156 -0.112 -0.047
2005-11-03 -0.115  1.271 -0.521 -0.099  1.050  0.501  0.154  0.332  0.573
2005-11-04 -0.324 -0.070 -0.113 -0.120  1.168  0.948  0.044  0.242  0.484
2005-11-07  0.131  0.621 -0.180  0.036  0.271  0.472  0.164  0.225  0.301
2005-11-08  0.054  0.033  0.210  0.233  0.035  0.085  0.109  0.096  0.083
2005-11-09 -0.255 -0.238 -0.190 -0.204  0.169  0.360 -0.137 -0.063  0.023
2005-11-10  0.100  0.092  0.103  0.144 -0.017  0.242  0.107  0.106  0.102
2005-11-11  0.062  1.333  0.046  0.065  0.735  1.071  0.317  0.475  0.682
2005-11-14  0.069 -0.469 -0.087 -0.070  0.001 -0.100 -0.039 -0.059 -0.095
```

Furthermore, to work with daily percentage returns, we have multiplied the series with 100.
Now let us plot the multivariate time series using the generic plot function for objects of class `timeSeries`

```
> plot(PensionFund[, -8])
```

Basic column statistics can easily be computed using the functions from the `colStats()` family of functions

LISTING 16.1: THE FAMILY OF FUNCTIONS TO COMPUTE COLUMN STATISTICAL PROPERTIES OF FINANCIAL AND ECONOMIC TIME SERIES DATA.

```
Function:
colStats            calculates column statistics,
colSums             calculates column sums,
colMeans            calculates column means,
colSds              calculates column standard deviations,
colVars             calculates column variances,
colSkewness         calculates column skewness,
```

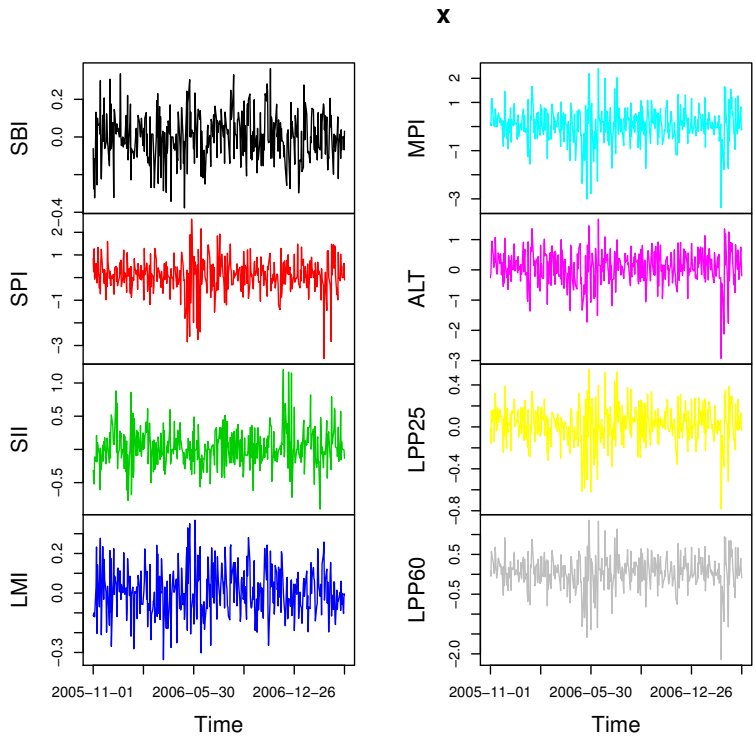

FIGURE 16.3: Time series plot of the Swiss pension fund benchmark Portfolio LPP2005. We have plotted the returns for the three Swiss assets, the three foreign assets, and for two of the benchmarks, the ones with the lowest and highest risk.

```
colKurtosis         calculates column kurtosis,
colMaxs             calculates maximum values in each column,
colMins             calculates minimum values in each column,
colProds            computes product of all values in each column,
colQuantiles        computes quantiles of each column.
```

Here we want to compute the the first two moment related statistics, the column means

```
> colMeans(PensionFund)
       SBI        SPI        SII        LMI        MPI        ALT      LPP25
4.0663e-05 8.4175e-02 2.3894e-02 5.5315e-03 5.9052e-02 8.5768e-02 2.3318e-02
     LPP40      LPP60
3.5406e-02 5.1073e-02
```

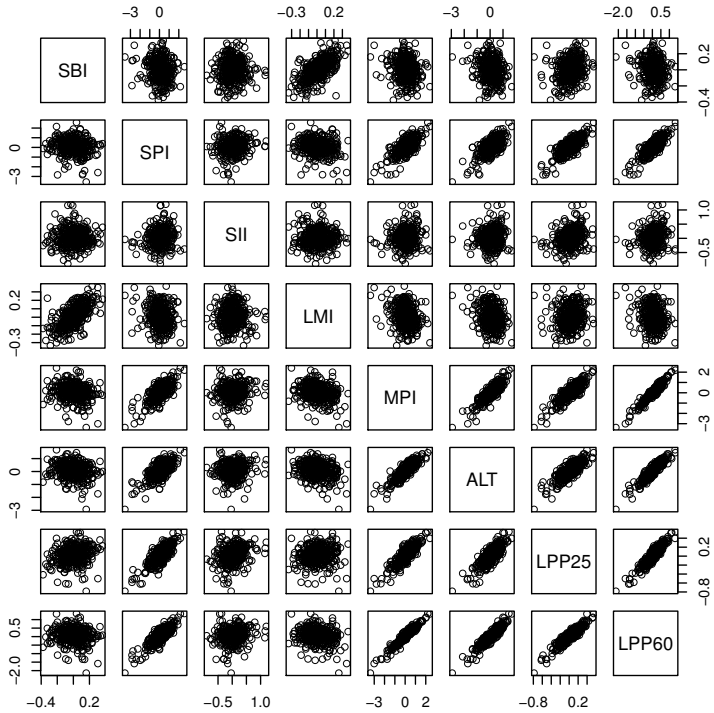

FIGURE 16.4: Correlation plot of the Swiss Pension fund benchmark Portfolio LPP2005. Shown are the six components of the portfolio.

the column standard deviations

```
> colSds(PensionFund)
    SBI     SPI     SII     LMI     MPI     ALT   LPP25   LPP40   LPP60
0.12609 0.76460 0.29178 0.12227 0.73146 0.56844 0.18067 0.28110 0.42343
```

Show the pairs correlations

```
> pairs(PensionFund, cex = 0.7, col = "steelblue", pch = 19)
```

16.2 STARS AND SEGMENTS PLOTS

The `stars()` plotting function is another method to visualize similarities in a multivariate data set. Each star plot or segment diagram represents

one row of the input x. Variables (columns) start on the right and wind counterclockwise around the circle. The size of the (scaled) column is shown by the distance from the centre to the point on the star or the radius of the segment representing the variable.

Example: World major stock market capitalization

For an exploratory data analysis of similarities we can plot a star graph

```
> stars(Caps, draw.segments = TRUE, labels = 2003:2008, ncol = 3)
```

The stars plot shows us the chronological development of the stock markets of the six years ranging from 2003 to 2008. The argument `draw.segments` was changed to `TRUE` to return a segment plot instead of the default star plot. The argument `labels` adds the years to the plot.
Finally let us decorate the plot with a main title and a marginal text with the source of the data.

```
> title(main = "Stock Market Capitalizations 2003 - 2008")
> mtext(side = 4, text = "Source: WEF Annual Report 2008", cex = 0.8,
     adj = 0, col = "darkgrey")
```

A second view to the data can be obtained if we transpose the data matrix. In this case we do not get displayed the development of the stock market capitalization over the years, instead we get the desired comparison of the different markets.

```
> stars(t(Caps), draw.segments = TRUE)
> title(main = "Capitalizations by Stock Market")
> mtext(side = 4, text = "Source: WEF Annual Report 2008", cex = 0.8,
     adj = 0, col = "darkgrey")
```

Example: Creating feature vectors

To compare the individual assets in the Swiss Pension Fund benchmark we can create feature vectors, for example of the basic distributional properties expressed by a box plot.

```
> x = as.data.frame(PensionFund)
> boxFeatures = boxplot(x, las = 2, col = topo.colors(9), pch = 19)
> abline(h = 0, lty = 3)
> title(main = "LPP Box Plot")
```

The returned value of the `boxplot()` function is a list with the $stats matrix entry amongst others. For the $stats matrix, each column contains the extreme of the lower whisker, the lower hinge, the median, the upper hinge and the extreme of the upper whisker for one group/plot.

```
> stars(t(boxFeatures$stats), labels = names(PensionFund), draw.segments = TRUE)
> title(main = "Boxplot Feature Vectors")
```

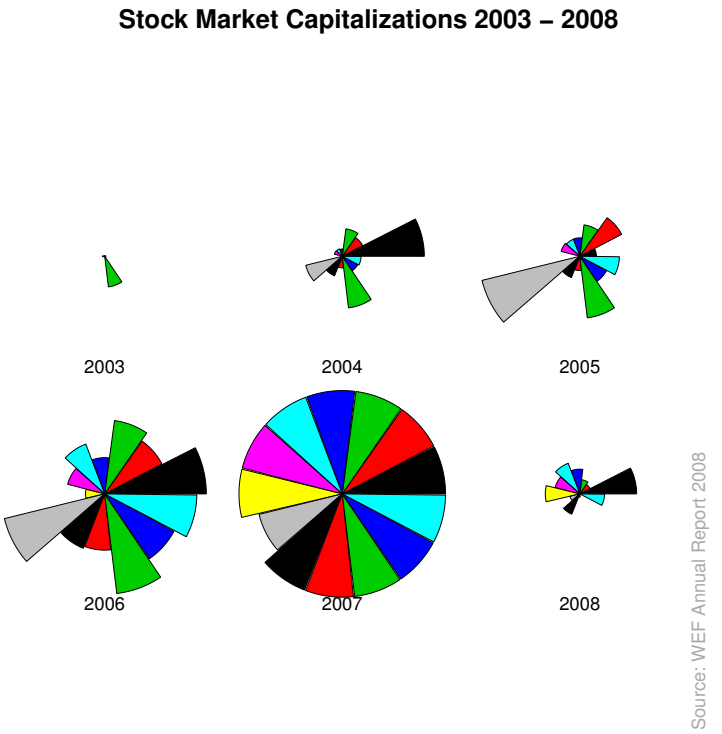

FIGURE 16.5: The stars plots show the growth of the market capitalization of the world major stock markets from 2003 to 2008. The plot shows impressively how the markets were raising up to 2007 and then collapsed 2008 during the subprime crises.

16.3 K-MEANS CLUSTERING

In statistics *k-means clustering* is a method of cluster analysis which aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean.

R's function `kmeans()` provides a variety of algorithms. The algorithm of Hartigan and Wong (1979) is used by default. Note that some authors use k-means to refer to a specific algorithm rather than the general method: most commonly the algorithm given by MacQueen (1967) but sometimes that given by Lloyd (1957) and Forgy (1965). The Hartigan-Wong algorithm generally does a better job than either of those, but trying several random starts is often recommended.

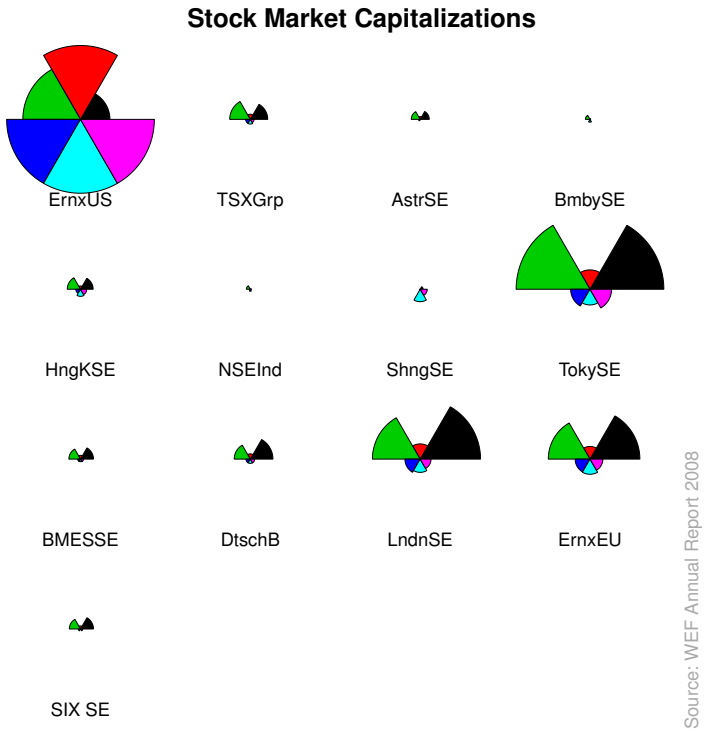

FIGURE 16.6: The stars plots compares the capitalizations of the world major stock markets. The 6 sectors of the stars show the temporal growth and fall from the year 2002 to 2008. Three groups of market can clearly be seen: The biggest market, Euronext US is dominant, Tokyo, London and Euronext EU form a second group, and the remaining stock markets fall in the third group.

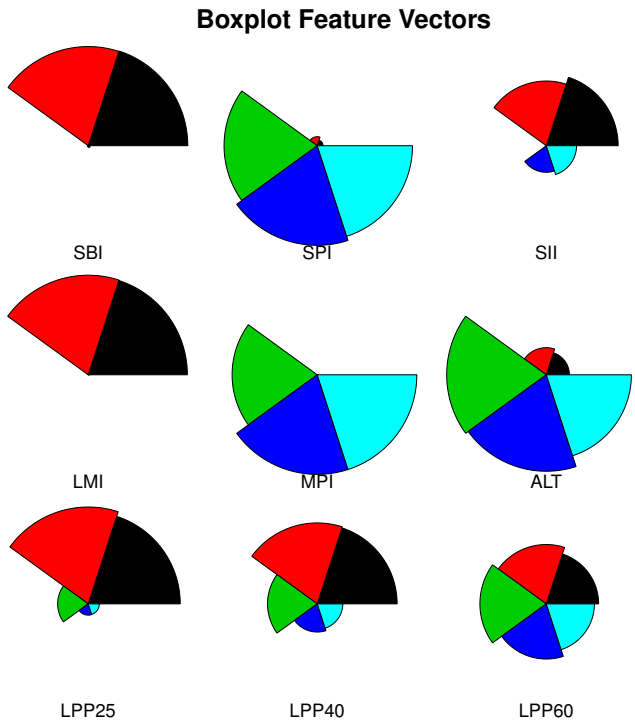

FIGURE 16.7: The stars plots compares the distributional properties of the financial returns of the pension fund portfolio, plotting the extreme of the lower whisker, the lower hinge, the median, the upper hinge and the extreme of the upper whisker for each asset class and the the three benchmarks.

Example: World major stock market capitalization

Which stock markets are similar?

```
> kmeans(t(Caps), centers = 3)
K-means clustering with 3 clusters of sizes 1, 9, 3

Cluster means:
     [,1]     [,2]   [,3]    [,4]    [,5]    [,6]
1 1328.95 12707.58 3632.3 15421.2 15650.8 9208.93
2  623.64   760.06  868.1  1243.9  2052.2  959.46
3 2496.52  2954.73 3446.0  4040.4  4135.1 2361.90

Clustering vector:
ErnxUS TSXGrp AstrSE BmbySE HngKSE NSEInd ShngSE TokySE BMESSE DtschB LndnSE
     1      2      2      2      2      2      2      3      2      2      3
ErnxEU SIX SE
     3      2

Within cluster sum of squares by cluster:
[1]       0 9014079 4491426

Available components:
[1] "cluster"  "centers"  "withinss" "size"
```

Note we observe the same three groups as expected from the star plot. To confirm this result inspect the clustering vector. The group named "2" contains the Euronext US stock exchange, the group named "3" contains the London, Tokyo and Euronext EU stock exchanges, and the remaining (smaller) stock exchanges are listed in group named "1".

Example: Pension fund benchmark portfolio

```
> features = t(boxFeatures$stats)
> rownames(features) = colnames(PensionFund)
> colnames(features) = c("lower whisker", "lower hinge", "median",
     "upper hinge", "upper whisker")
> kmeans(features, center = 3)
K-means clustering with 3 clusters of sizes 3, 3, 3

Cluster means:
  lower whisker lower hinge   median upper hinge upper whisker
1      -0.66441   -0.143260 0.041890     0.23043       0.75932
2      -1.34477   -0.265176 0.100851     0.48579       1.57098
3      -0.32829   -0.077806 0.010597     0.09635       0.34018

Clustering vector:
  SBI   SPI   SII   LMI   MPI   ALT LPP25 LPP40 LPP60
    3     2     1     3     2     2     3     1     1

Within cluster sum of squares by cluster:
[1] 0.121471 0.062491 0.020388
```

```
Available components:
[1] "cluster"  "centers"  "withinss" "size"
```

The clustering of the box plot features are grouped in the following sense: Group "3" contains the low risk assets SMI, LMI and LPP25, Group "2" ... Check the result, I am (DW) not satisfied with it ...

16.4 HIERARCHICAL CLUSTERING

In statistics hierarchical clustering is a method of cluster analysis which seeks to build a hierarchy of clusters. Strategies for hierarchical clustering generally fall into two types: (i) *Agglomerative*, this is a "bottom up" approach where each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy ans (ii) *Divisive*, this is a "top down" approach where all observations start in one cluster, and splits are performed recursively as one moves down the hierarchy. In general, the merges and splits are determined in a greedy manner. The results of hierarchical clustering are usually presented in a hierarchical tree graph named dendrogram.
Dissimilarity Measure: In order to decide which clusters should be combined (for agglomerative), or where a cluster should be split (for divisive), a measure of dissimilarity between sets of observations is required. In most methods of hierarchical clustering, this is achieved by use of an appropriate metric (a measure of distance between pairs of observations), and a linkage criteria which specifies the dissimilarity of sets as a function of the pairwise distances of observations in the sets.
Metric: The choice of an appropriate metric will influence the shape of the clusters, as some elements may be close to one another according to one distance and farther away according to another. Some commonly used metrics for hierarchical clustering include: (i) Euclidean distance, (ii) maximum distance, (iii) Manhattan distance, (iv) Canberra distance, (v) binary distance. (vi) Minkowski distance. For details we refer to R's help page for the function `dist()` which computes and returns the distance matrix for a given data set.
Linkage:
R has several functions for hierarchical clustering, see the CRAN Task View *Cluster Analysis and Finite Mixture Models* for more information.

Example: Major world stock market capitalization

Now we investigate the question how are the stock markets related performing a hierarchical clustering.

```
> Dist = dist(t(Caps), method = "manhattan")
> class(Dist)
```

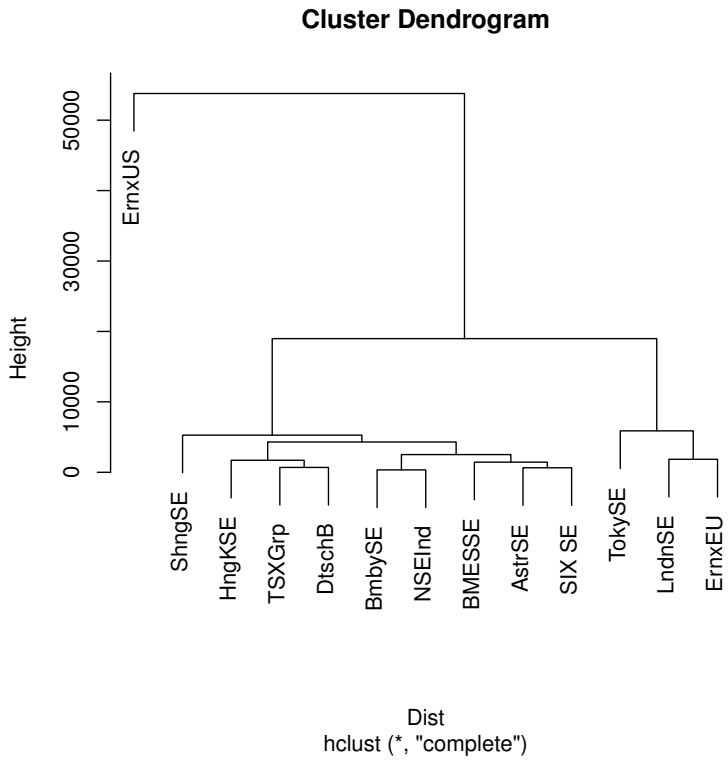

FIGURE 16.8: A dendrogram plot from clustering major world stock market capitalization.

```
[1] "dist"

> Dist

         ErnxUS   TSXGrp   AstrSE   BmbySE   HngKSE   NSEInd   ShngSE   TokySE
TSXGrp 49480.68
AstrSE 52705.88  3225.19
BmbySE 53446.53  3965.85  1782.22
HngKSE 49620.57  1694.75  3085.30  3825.96
NSEInd 53783.13  4302.45  1800.82   336.60  4162.56
ShngSE 50951.95  5270.67  4521.10  3172.36  3604.41  3388.67
TokySE 39934.79 14675.38 17900.57 18641.23 14815.27 18977.83 16146.65
BMESSE 51270.54  1789.86  1435.33  2251.92  1831.68  2512.59  4461.84 16465.24
DtschB 49601.73   690.01  3104.14  3844.80  1708.34  4181.40  5158.07 14796.43
LndnSE 42314.33  9428.57 12653.77 13394.42  9568.46 13731.02 10899.84  5246.81
ErnxEU 42183.10  8792.49 12017.69 12758.34  8932.38 13094.95 10263.76  5882.89
SIX SE 52120.51  2639.83   639.90  2422.12  2524.96  2440.72  4814.13 17315.21
         BMESSE   DtschB   LndnSE   ErnxEU
```

```
TSXGrp
AstrSE
BmbySE
HngKSE
NSEInd
ShngSE
TokySE
BMESSE
DtschB  1668.81
LndnSE 11218.43  9549.62
ErnxEU 10582.35  8913.55  1845.21
SIX SE   851.69  2518.78 12068.40 11432.33
```

```
> Clust = hclust(Dist, method = "complete")
> plot(Clust)
```

Again, the result is the same.

Example: Pension fund benchmark portfolio

```
> X = t(getDataPart(PensionFund))
> Dist = dist(X, "euclidean")
> Clust = hclust(Dist, "complete")
> plot(Clust)
> box()
```

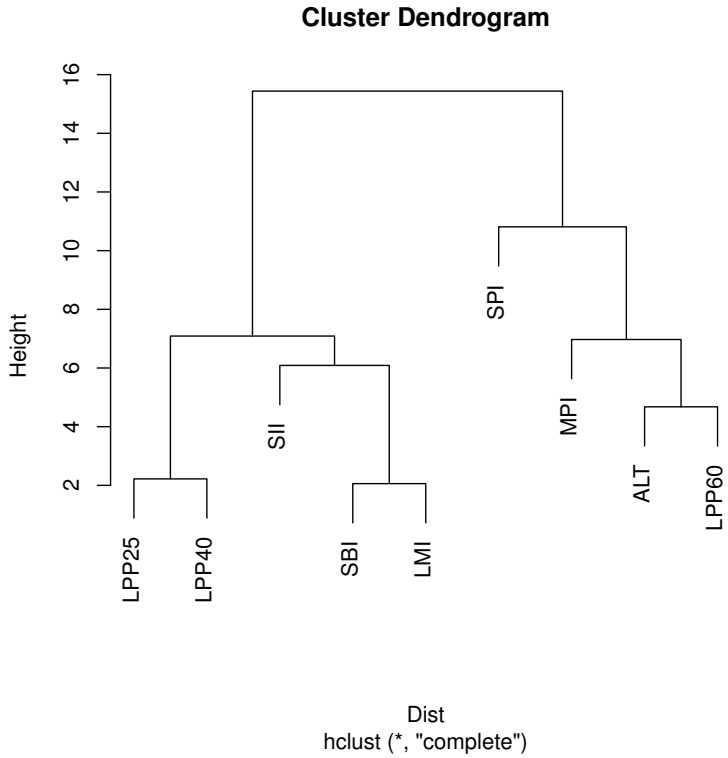

FIGURE 16.9: A dendrogram plot from clustering Swiss pension fund portfolio.

PART V

CASE STUDIES: UTILITY FUNCTIONS

CHAPTER 17

COMPUTE SKEWNESS STATISTICS

17.1 ASSIGNMENT

The basic R environment has no function to compute the skewness statistics for a data set. Let us add one in the style of the functions `mean()` and `var()`.
There exist several flavours to compute the skewness of a numerical vector. Two of these are the 'moment' or 'fisher' methods. Our definitions follow the implementation of the skewness function in SPlus: The `"moment"` forms are based on the definitions of skewness and kurtosis for distributions; these forms should be used when resampling (bootstrap or jackknife). The `"fisher"` forms correspond to the usual 'unbiased' definition of sample variance, though in the case of skewness exact unbiasedness is not possible (SPlus 2009).

17.2 R IMPLEMENTATION

First let us define an S3 method for the `skewness`

```
> skewness <- function(x, ...) {
      UseMethod("skewness")
  }
```

The default method is thought to handle numeric objects. We formulate three arguments, the numerical object `x`, a logical flag `na.rm` to decide if missing values should be removed, and the name of the statistical `method` to how to compute the skewness.
The function contains the following steps:

1. check input object
2. check for valid method
3. remove optionally NAs

4. transform integers to numeric values if required

5. compute the skewness for the desired method

6. add the name of the method as an attribute to the final result.

```
> skewness.default <-
 function (x, na.rm = FALSE, method = c("moment", "fisher"), ...)
 {
     # Arguments:
     #   x - any numerical object
     #   na.rm - logical flag, should missing values be removed?
     #   method - character string specifying the computation method.
     #       "fisher" for Fisher's g1 skewness version
     #       "moment" for the functional forms of the statistics
     #   ... - not used

     # 1 Check Input Object x:
     if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
         warning("argument is not numeric or logical: returning NA")
         return(as.numeric(NA))}

     # 2 Check Method:
     method <- match.arg(method)

     # 3 Remove NAs:
     if (na.rm) x = x[!is.na(x)]

     # 4 Transform to Numeric:
     n = length(x)
     if (is.integer(x)) x = as.numeric(x)

     # 5 Compute Selected Method:
     if (method == "moment") {
         skewness = sum((x-mean(x))^3/sqrt(var(x))^3)/length(x)
     }
     if (method == "fisher") {
         if (n < 3)
             skewness = NA
         else
             skewness = ((sqrt(n*(n-1))/(n-2))*(sum(x^3)/n))/((sum(x^2)/n)^(3/2))
     }

     # 6 Add Control Attribute:
     attr(skewness, "method") <- method

     # Return Value:
     skewness
 }
```

17.3 EXAMPLES

As an example, let us compute the skewness for a vector of 100 standardized normal variables.

```
> set.seed(1943)
> skewness(rnorm(100))
[1] -0.023230
attr(,"method")
[1] "moment"
```

CHAPTER 18

COMPUTE KURTOSIS STATISTICS

18.1 ASSIGNMENT

As in the case of the `skewness()` function, the basic R environment has no function to compute the kurtosis statistics for a data set. Let us add one in the style of the function `skewness()`.
As in the previous example we follow the implementation of the kurtosis in the SPlus environment, 2009. The kurtosis function in SPlus allows for three different choices of computations, `"excess"`, `"moment"`, and `"fisher"`. The first method stands for the excess kurtosis, the second for the moments statistics and the third for Fisher's $g2$ kurtosis version.

18.2 R IMPLEMENTATION

Following the procedure of the previous case study, let us define the S3 method for the `kurtosis()` function

```
> kurtosis <- function(x, ...) {
    UseMethod("kurtosis")
 }
```

and then implement the default method for numeric objects.

```
> kurtosis.default <-
 function (x, na.rm=FALSE, method=c("excess", "moment", "fisher"), ...)
 {
    # Arguments:
    #   x - any numerical object
    #   na.rm - logical flag, should missing values be removed?
    #   method - character string specifying the computation method.
    #       "fisher" for Fisher's g2 kurtosis version
    #       "moment" for the functional forms of the statistics
    #       "excess" for the excess kurtosis

    # 1 Check Input Object x:
    if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
```

```
        warning("argument is not numeric or logical: returning NA")
        return(as.numeric(NA))}

    # 2 Check Method:
    method = match.arg(method)

    # 3 Remove NAs:
    if (na.rm) x = x[!is.na(x)]

    # 4 Transform to Numeric:
    n = length(x)
    if (is.integer(x)) x = as.numeric(x)

    # 5 Compute Selected Method:
    if (method == "excess") {
        kurtosis = sum((x-mean(x))^4/var(x)^2)/length(x) - 3
    }
    if (method == "moment") {
        kurtosis = sum((x-mean(x))^4/var(x)^2)/length(x)
    }
    if (method == "fisher") {
        kurtosis = ((n+1)*(n-1)*((sum(x^4)/n)/(sum(x^2)/n)^2 -
            (3*(n-1))/(n+1)))/((n-2)*(n-3))
    }

    # 6 Add Control Attribute:
    attr(kurtosis, "method") <- method

    # Return Value:
    kurtosis
}
```

18.3 EXAMPLES

Compute the kurtosis for a sample of 100 normal random numbers with mean zero and variance one

```
> set.seed(4711)
> x <- rnorm(1000)
> kurtosis(x)
[1] -0.068852
attr(,"method")
[1] "excess"
```

Now compute the moment kurtosis

```
> kurtosis(x, method = "moment")
[1] 2.9311
attr(,"method")
[1] "moment"
```

CHAPTER 19

EXTRACTING PACKAGE DESCRIPTION

19.1 ASSIGNMENT

Write an R function which extracts the DESCRIPTION file from a desired package.

19.2 R IMPLEMENTATION

We compose the command as a text string, parse the text and evaluate the the command. As an example for a typical R package we use the `utiles` package.

```
> package <- "utils"
> cmd <- paste("library(help=", package, ")", sep = "")
> ans <- eval(parse(text = cmd))
```

The `library()` returns a list with three elements, where the last element named `info` is by itself an unnamed list with three elements,

```
> names(ans)
[1] "name" "path" "info"
> length(ans$info)
[1] 3
```

The first contains the description information

```
> ans$info[[1]]
[1] "Package:       utils"
[2] "Version:       2.11.1"
[3] "Priority:      base"
[4] "Title:         The R Utils Package"
[5] "Author:        R Development Core Team and contributors worldwide"
[6] "Maintainer:    R Core Team <R-core@r-project.org>"
[7] "Description:   R utility functions"
[8] "License:       Part of R 2.11.1"
[9] "Built:         R 2.11.1; ; 2010-05-31 14:42:44 UTC; unix"
```

Now let us put everything together and write the R function `listDescription()`

```
> listDescription <-
 function(package)
 {
     # Arguments:
     #   package - a character, the name of the package

     # Extract Description:
     cmd = paste("library(help =", package, ")", sep = "" )
     ans = eval(parse(text = cmd))
     description = ans$info[[1]]

     # Return Value:
     cat("\n", package, "Description:\n\n")
     cat(paste(" ", description), sep = "\n")
     invisible()
 }
```

Note that the second list entry `ans$info[[1]]` contains the index information. As an exercise write a function `listIndex()` which extracts the index information.

19.3 EXAMPLES

Here comes an example how to use the function `listDescription()`

```
> listDescription("utils")
 utils Description:

  Package:       utils
  Version:       2.11.1
  Priority:      base
  Title:         The R Utils Package
  Author:        R Development Core Team and contributors worldwide
  Maintainer:    R Core Team <R-core@r-project.org>
  Description:   R utility functions
  License:       Part of R 2.11.1
  Built:         R 2.11.1; ; 2010-05-31 14:42:44 UTC; unix
```

CHAPTER 20

FUNCTION LISTING AND COUNTING

20.1 ASSIGNMENT

Write R functions which list the function names and the number of functions in a given R package.

20.2 R IMPLEMENTATION

Let us start to write a function which lists all functions by name in an R package. First we check if the package is loaded calling the function `require()`. Note that `require()` returns (invisbly) a logical indicating whether the required package is available. If the package is loaded we list the functions

```
> listFunctions <-
 function(package)
 {
     # Arguments:
     #   package - a character string, the name of the Package

     # Listing - Original code borrowed from B. Ripley:

     # 1 Package loaded?
     loaded <- require(package, character.only = TRUE, quietly = TRUE)

     # 2 Function listing, if package was loaded:
     if(loaded) {
         # List Names:
         env <- paste("package", package, sep = ":")
         nm <- ls(env, all = TRUE)
         ans = nm[unlist(lapply(nm, function(n) exists(n, where = env,
             mode = "function", inherits = FALSE)))]
     } else {
         ans = character(0)
     }
```

```
    # Return Value:
    ans
}
```

Now let us write a function to count the number of functions in a given package. Here use the previous function listing and just compute the length of the returned vector.

```
> countFunctions <-
 function(package)
 {
    # Arguments:
    #   package - a character string, the name of the Package

    # Count Functions:
    ans = length(listFunctions(package))
    names(ans) = package

    # Return Value:
    ans
 }
```

20.3 EXAMPLES

List all functions in the `utils` package by name.

```
> listFunctions("utils")
  [1] "?"                          ".DollarNames"
  [3] "alarm"                      "apropos"
  [5] "argsAnywhere"               "as.person"
  [7] "as.personList"              "as.relistable"
  [9] "as.roman"                   "aspell"
 [11] "assignInNamespace"          "available.packages"
 [13] "browseEnv"                  "browseURL"
 [15] "browseVignettes"            "bug.report"
 [17] "capture.output"             "checkCRAN"
 [19] "chooseBioCmirror"           "chooseCRANmirror"
 [21] "citation"                   "citEntry"
 [23] "citFooter"                  "citHeader"
 [25] "close.socket"               "combn"
 [27] "compareVersion"             "contrib.url"
 [29] "count.fields"               "CRAN.packages"
 [31] "data"                       "data.entry"
 [33] "dataentry"                  "de"
 [35] "de.ncols"                   "de.restore"
 [37] "de.setup"                   "debugger"
 [39] "demo"                       "download.file"
 [41] "download.packages"          "dump.frames"
 [43] "edit"                       "emacs"
 [45] "example"                    "file_test"
 [47] "file.edit"                  "find"
 [49] "findLineNum"                "fix"
 [51] "fixInNamespace"             "flush.console"
```

```
 [53] "formatOL"                   "formatUL"
 [55] "getAnywhere"                "getCRANmirrors"
 [57] "getFromNamespace"           "getS3method"
 [59] "getTxtProgressBar"          "glob2rx"
 [61] "head"                       "head.matrix"
 [63] "help"                       "help.request"
 [65] "help.search"                "help.start"
 [67] "history"                    "install.packages"
 [69] "installed.packages"         "is.relistable"
 [71] "limitedLabels"              "loadhistory"
 [73] "localeToCharset"            "ls.str"
 [75] "lsf.str"                    "maintainer"
 [77] "make.packages.html"         "make.socket"
 [79] "makeRweaveLatexCodeRunner" "memory.limit"
 [81] "memory.size"                "menu"
 [83] "methods"                    "mirror2html"
 [85] "modifyList"                 "new.packages"
 [87] "news"                       "normalizePath"
 [89] "nsl"                        "object.size"
 [91] "old.packages"               "package.contents"
 [93] "package.skeleton"           "packageDescription"
 [95] "packageStatus"              "page"
 [97] "person"                     "personList"
 [99] "pico"                       "prompt"
[101] "promptData"                 "promptPackage"
[103] "rc.getOption"               "rc.options"
[105] "rc.settings"                "rc.status"
[107] "read.csv"                   "read.csv2"
[109] "read.delim"                 "read.delim2"
[111] "read.DIF"                   "read.fortran"
[113] "read.fwf"                   "read.socket"
[115] "read.table"                 "readCitationFile"
[117] "recover"                    "relist"
[119] "remove.packages"            "Rprof"
[121] "Rprofmem"                   "RShowDoc"
[123] "RSiteSearch"                "rtags"
[125] "Rtangle"                    "RtangleSetup"
[127] "RtangleWritedoc"            "RweaveChunkPrefix"
[129] "RweaveEvalWithOpt"          "RweaveLatex"
[131] "RweaveLatexFinish"          "RweaveLatexOptions"
[133] "RweaveLatexSetup"           "RweaveLatexWritedoc"
[135] "RweaveTryStop"              "savehistory"
[137] "select.list"                "sessionInfo"
[139] "setBreakpoint"              "setRepositories"
[141] "setTxtProgressBar"          "stack"
[143] "Stangle"                    "str"
[145] "strOptions"                 "summaryRprof"
[147] "Sweave"                     "SweaveHooks"
[149] "SweaveSyntConv"             "tail"
[151] "tail.matrix"                "tar"
[153] "timestamp"                  "toBibtex"
[155] "toLatex"                    "txtProgressBar"
[157] "type.convert"               "unstack"
[159] "untar"                      "unzip"
[161] "update.packages"            "update.packageStatus"
```

```
[163] "upgrade"                 "url.show"
[165] "URLdecode"               "URLencode"
[167] "vi"                      "View"
[169] "vignette"                "write.csv"
[171] "write.csv2"              "write.socket"
[173] "write.table"             "wsbrowser"
[175] "xedit"                   "xemacs"
[177] "zip.file.extract"
```

How many functions are in the `utils` package?

```
> countFunctions("utils")
utils
  177
```

PART VI

CASE STUDIES: ASSET MANAGEMENT

CHAPTER 21

GENERALIZED ERROR DISTRIBUTION

21.1 ASSIGNMENT

Write R functions for the *Generalized Error Distribution*, GED. Nelson [1991] introduced the *Generalized Error Distribution* for modeling GARCH time series processes. The GED has exponentially stretched tails. The GED includes the normal distribution as a special case, along with many other distributions. Some are more fat tailed than the normal, for example the double exponential, and others are more thin-tailed like the uniform distribution.
Write R functions to compute density, probabilities and quantiles for the GED. Write an R function to generate random variates. Write an R function to estimate the parameters for a GED using the maximum log-likelihood approach.

References

Daniel B. Nelson, 1991,
Conditional Heteroskedasticity in Asset Returns: A New Approach Econometrica 59, 347–370

Wikipedia, Generalized Normal Distribution, 2010,
http://en.wikipedia.org/wiki/Generalized_normal_distribution

21.2 R IMPLEMENTATION

The density of the standardized GED is described by the following formula

$$f(x) = \frac{v\,exp[-\frac{1}{2}|z/\lambda^v|]}{\lambda 2^{(1+1/v)}\Gamma(1/v)} \tag{21.1}$$

where $\Gamma()$ is the gamma function, and

$$\lambda \equiv [2^{-2/\nu}\Gamma(1/\nu)/\Gamma(3/\nu)]^{1/2} \tag{21.2}$$

ν is a tail-thickness parameter. When $\nu = 2$, x has a standard normal distribution. For $\nu < 2$, the distribution of x has thicker tails than the normal. For example when $\nu = 1$, x has a double exponential distribution. For $\nu > 2$, the distribution has thinner tails than the normal, for $\nu = \infty$, x is uniformly distributed on the interval $[-\sqrt{3}, \sqrt{3}]$.

GED density function

Write an R function to compute the density for the GED

```
> dged <- function(x, mean = 0, sd = 1, nu = 2) {
     z = (x - mean)/sd
     lambda = sqrt(2^(-2/nu) * gamma(1/nu)/gamma(3/nu))
     g = nu/(lambda * (2^(1 + 1/nu)) * gamma(1/nu))
     density = g * exp(-0.5 * (abs(z/lambda))^nu)/sd
     density
 }
```

GED probability function

Write an R function to compute the probability function for the GED.

```
> pged <- function(q, mean = 0, sd = 1, nu = 2) {
     q = (q - mean)/sd
     lambda = sqrt(2^(-2/nu) * gamma(1/nu)/gamma(3/nu))
     g = nu/(lambda * (2^(1 + 1/nu)) * gamma(1/nu))
     h = 2^(1/nu) * lambda * g * gamma(1/nu)/nu
     s = 0.5 * (abs(q)/lambda)^nu
     probability = 0.5 + sign(q) * h * pgamma(s, 1/nu)
     probability
 }
```

GED quantile function

Write an R function to compute the quantile function for the GED.

```
> qged <- function(p, mean = 0, sd = 1, nu = 2) {
     lambda = sqrt(2^(-2/nu) * gamma(1/nu)/gamma(3/nu))
     q = lambda * (2 * qgamma((abs(2 * p - 1)), 1/nu))^(1/nu)
     quantiles = q * sign(2 * p - 1) * sd + mean
     quantiles
 }
```

GED random number generation

Write an R function to generate random variates from the GED.

```
> rged <- function(n, mean = 0, sd = 1, nu = 2) {
    lambda = sqrt(2^(-2/nu) * gamma(1/nu)/gamma(3/nu))
    r = rgamma(n, 1/nu)
    z = lambda * (2 * r)^(1/nu) * sign(runif(n) - 1/2)
    rvs = z * sd + mean
    rvs
}
```

GED parameter estimation

Write an R function to estimate the parameters from empirical return series data for a GED using the maximum log-likelihood approach.

```
> gedFit <- function(x, ...) {
    start = c(mean = mean(x), sd = sqrt(var(x)), nu = 2)
    loglik = function(x, y = x) {
        f = -sum(log(dged(y, x[1], x[2], x[3])))
        f
    }
    fit = nlminb(start = start, objective = loglik, lower = c(-Inf,
        0, 0), upper = c(Inf, Inf, Inf), y = x, ...)
    names(fit$par) = c("mean", "sd", "nu")
    fit
}
```

21.3 EXAMPLES

Plot GED density

Compute and display the GED Density with zero mean and unit variance in the range (-4,4) for three different parameter settings of ν, 1, 2, and 3.

```
> x = seq(-4, 4, length = 501)
> y = dged(x, mean = 0, sd = 1, nu = 1)
> plot(x, y, type = "l", , col = "blue", main = "GED", ylab = "Density",
    xlab = "")
> y = dged(x, mean = 0, sd = 1, nu = 2)
> lines(x, y, col = "red")
> y = dged(x, mean = 0, sd = 1, nu = 3)
> lines(x, y, col = "green")
> abline(h = 0, col = "grey")
```

Normalization of the GED density

Show for a random parameter setting of the mean and standard deviation that the GED density is normalized.

```
> set.seed(4711)
> MEAN = rnorm(1)
> SD = runif(1)
> for (NU in 1:3) print(integrate(dged, -Inf, Inf, mean = MEAN,
```

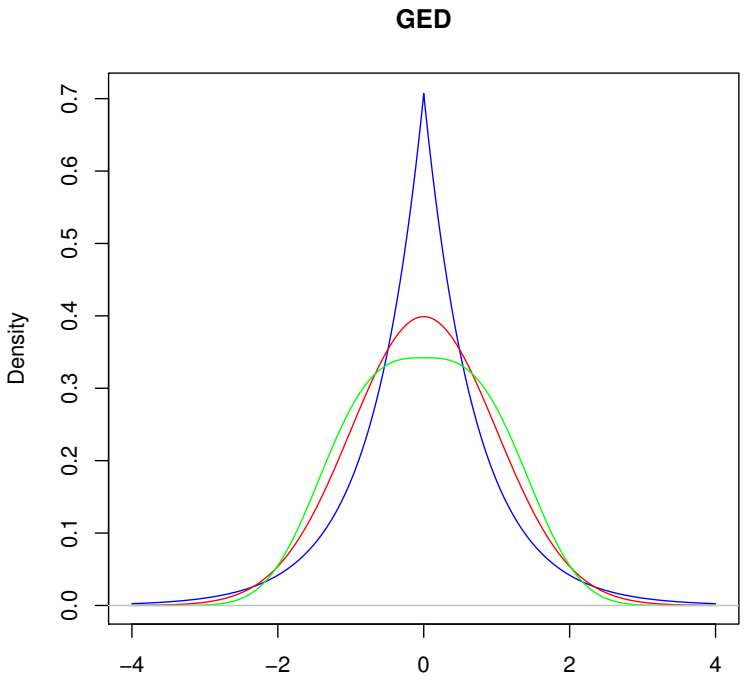

FIGURE 21.1: GED Density Plots for nu=c(1, 2, 3)

```
    sd = SD, nu = NU))
1 with absolute error < 0.00012
1 with absolute error < 5.5e-07
1 with absolute error < 5.6e-05
```

Repeat this computation for other settings of `MEAN` and `SD`.

Check the pged and qged functions

To check these functions, here for zero mean and unit standard deviation, we compute the quantiles from the probabilities of quantiles.

```
> q = c(0.001, 0.01, 0.1, 0.5, 0.9, 0.99, 0.999)
> q
[1] 0.001 0.010 0.100 0.500 0.900 0.990 0.999
> p = pged(q)
> p
[1] 0.50040 0.50399 0.53983 0.69146 0.81594 0.83891 0.84110
```

```
> qged(p)
[1] 0.001 0.010 0.100 0.500 0.900 0.990 0.999
```

LPP histogram plot

We want to test the three Swiss Pension Fund Indices if they are normal distributed. These are Pictet's so calle LPP indices named LPP25, LPP40, and LPP60. The numbers reflect the amount of equities included, thus the indices reflect benchmarks with increasing risk levels.
The data set is downloadable from the r-forge web site as a semicolon separated csv file. The file has 10 columns, the first holds the dates, the next 6 index data of bond, reit and stock indices, and the last tree the Swiss pension fund index benchmarks.
Download the data and select columns 2 to 4

```
> library(fEcofin)
> data(SWXLP)
> LPP.INDEX <- SWXLP[, 5:7]
> head(LPP.INDEX)
   LP25  LP40  LP60
1 99.81 99.71 99.55
2 98.62 97.93 96.98
3 98.26 97.36 96.11
4 98.13 97.20 95.88
5 98.89 98.34 97.53
6 99.19 98.79 98.21
```

Compute daily percentual returns

```
> LPP = 100 * diff(log(as.matrix(LPP.INDEX)))
```

Create a nice histogram plot which adds a normal distribution fit to the histogram, adds the mean as a vertical Line, and adds rugs to the x-axis. Use nice colors to display the histogram.

```
> histPlot <- function(x, ...) {
    X = as.vector(x)
    H = hist(x = X, ...)
    box()
    grid()
    abline(h = 0, col = "grey")
    mean = mean(X)
    sd = sd(X)
    xlim = range(H$breaks)
    s = seq(xlim[1], xlim[2], length = 201)
    lines(s, dnorm(s, mean, sd), lwd = 2, col = "brown")
    abline(v = mean, lwd = 2, col = "orange")
    Text = paste("Mean:", signif(mean, 3))
    mtext(Text, side = 4, adj = 0, col = "darkgrey", cex = 0.7)
    rug(X, ticksize = 0.01, quiet = TRUE)
    invisible(s)
}
```

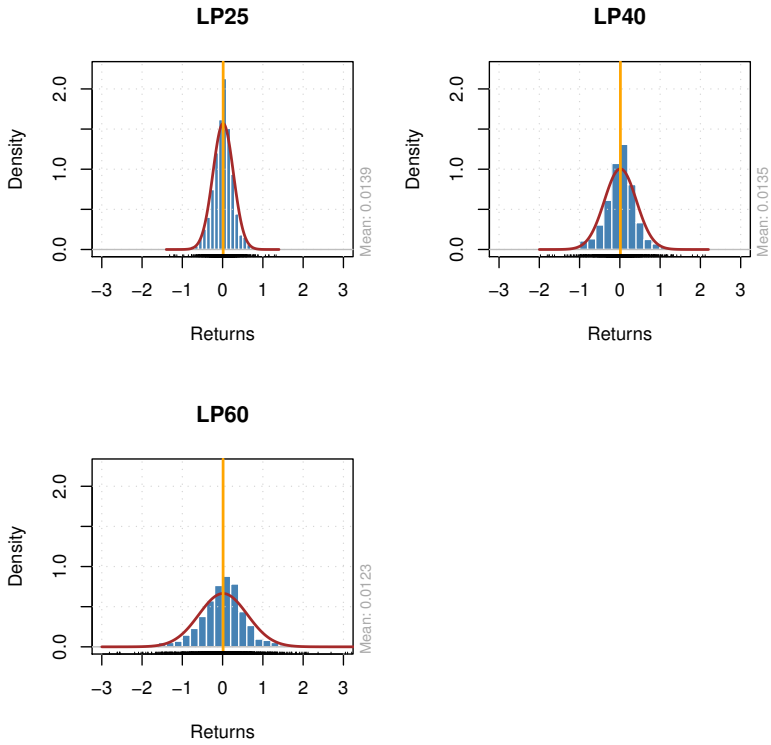

FIGURE 21.2: Histogram Plots for the LPP Benchmark Indices

Plot the three histograms

```
> par(mfrow = c(2, 2))
> main = colnames(LPP)
> for (i in 1:3) histPlot(LPP[, i], main = main[i], col = "steelblue",
      border = "white", nclass = 25, freq = FALSE, xlab = "Returns")
```

Parameter estimation

Fit the parameters to a GED for the three LPP benchmark series

```
> param = NULL
> for (i in 1:3) param = rbind(param, gedFit(LPP[, i])$par)
> rownames(param) = colnames(LPP)
> param
          mean      sd     nu
LP25 0.023525 0.25266 1.1693
LP40 0.032626 0.39439 1.1178
LP60 0.040193 0.59533 1.1053
```

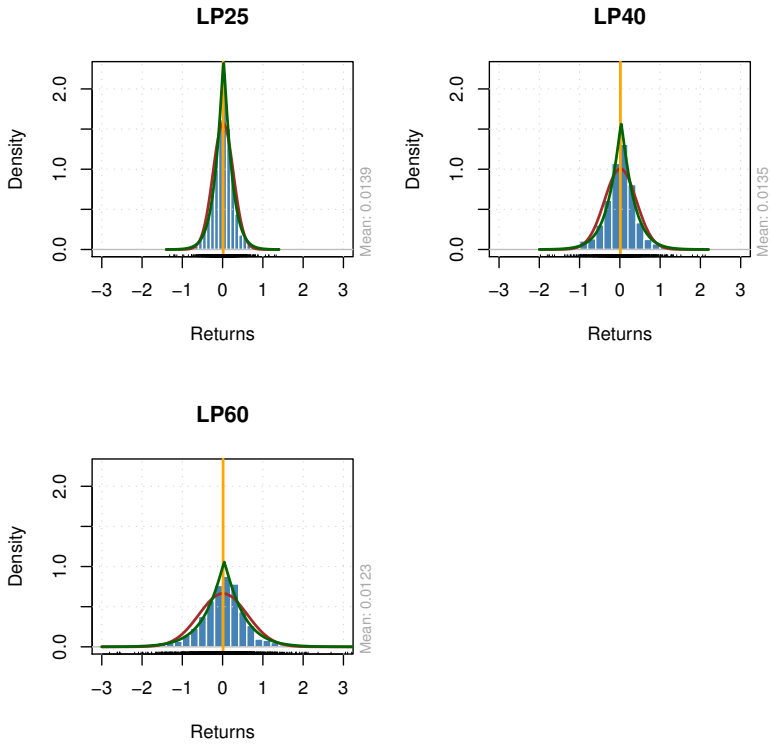

FIGURE 21.3: LPP Histogram Plots with fitted Normal (brown) and GED (green). Note that all three histogram plots are on the same scale.

Overlay the histograms with a fitted GED

```
> par(mfrow = c(2, 2))
> main = colnames(LPP)
> for (i in 1:3) {
      u = histPlot(LPP[, i], main = main[i], col = "steelblue",
          border = "white", nclass = 25, freq = FALSE, xlab = "Returns")
      v = dged(u, mean = param[i, 1], sd = param[i, 2], nu = param[i,
          3])
      lines(u, v, col = "darkgreen", lwd = 2)
  }
```

21.4 EXERCISES

Rewrite the functions `dged()` as in the case of `dnorm()`

```
> args(dnorm)
```

```
function (x, mean = 0, sd = 1, log = FALSE)
NULL
```

with an additional argument of `log`. Use this function for the estimation of the distributional parameters in the function `gedFit()`.
Rewrite the functions `pged()` and `qged()` as in the case of `pnorm()` and `qnorm()`

```
> args(pnorm)
function (q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
NULL
> args(qnorm)
function (p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
NULL
```

with additional arguments of `lower.tail` and `log.p`.
The arguments `log` and `log.p` are logical values, if `TRUE` probabilities `p` are given as `log(p)`. `lower.tail` is a logical value; if TRUE, probabilities are $P[X \leq x]$, otherwise, $P[X > x]$.

CHAPTER 22

SKEWED RETURN DISTRIBUTIONS

22.1 ASSIGNMENT

Fernandez and Steel 1996, showed a general method to transform an unimodal symmetric distribution into a skew symmetric distribution. Use this approach and write function for the density, probabilities and quantiles for the skew Normal distribution.

References

Fernandez and Steel, 1996,
On Bayesian Modeling of Fat Tails and Skewness,
Tilburg University, Center for Economic Research,
Discussion Paper Series Number 1996-58

22.2 R IMPLEMENTATION

Consider a univariate pdf $f(\cdot)$ which is unimodal and symmetric around 0. Fernandez and Stell then generates the following class of skewed distributions

$$p(\varepsilon|\gamma) = \frac{2}{\gamma + \frac{1}{\gamma}} \{ f(\frac{\varepsilon}{\gamma}) I_{[0,\infty)}(\varepsilon) + f(\gamma\varepsilon) I_{-\infty,0)}(\varepsilon) \} \tag{22.1}$$

Their basic idea is the introduction of inverse scale factors in the positive and negative orthant.

Skewed normal density

First we write a function for the standardized skew density function with mean zero and unit variance, named `.dsnorm()`

```
> .dsnorm <- function(x, lambda)
 {
     # Standardize x:
     absMoment = 2/sqrt(2*pi)
     mu = absMoment * (lambda - 1/lambda)
     sigma <- sqrt((1-absMoment^2)*(lambda^2+1/lambda^2) + 2*absMoment^2 - 1)
     z <- x*sigma + mu

     # Compute Density:
     Lambda <- lambda^sign(z)
     g <- 2 / (lambda + 1/lambda)
     Density <- g * sigma * dnorm(x = z/Lambda)

     # Return Value:
     Density
 }
```

Then we generalize the density function for arbitrary mean and variance

```
> dsnorm <- function(x, mean = 0, sd = 1, lambda = 1)
 {
     # Shift and Scale:
     Density <- .dsnorm(x = (x-mean)/sd, lambda = lambda) / sd

     # Return Value:
     Density
 }
```

Skewed normal probability

Here are the functions for the probabilities

```
> .psnorm <- function(q, lambda)
 {
     # Standardize x:
     absMoment <- 2/sqrt(2*pi)
     mu <- absMoment * (lambda - 1/lambda)
     sigma <- sqrt((1-absMoment^2)*(lambda^2+1/lambda^2) + 2*absMoment^2 - 1)
     z <- q*sigma + mu

     # Compute Probabilities:
     Lambda <- lambda^sign(z)
     g <- 2  / (lambda + 1/lambda)
     Probabilities <- Heaviside(z) - sign(z) * g * Lambda * pnorm(q = -abs(z)/Lambda)

     # Return Value:
     Probabilities
 }
```

where `Heaviside()` implements the Heaviside function

```
> Heaviside <- function(x, a = 0)
 {
     # Compute Heaviside Function:
     heaviside <- (sign(x-a) + 1)/2
```

```
    # Return Value:
    heaviside
}
```

Then

```
> psnorm <- function(q, mean = 0, sd = 1, lambda = 1)
 {
    # Shift and Scale:
    Probabilities <- .psnorm(q = (q-mean)/sd, lambda = lambda)

    # Return Value:
    Probabilities
 }
```

22.3 EXAMPLES

Plot the skew normal density for the following values of $\lambda = \{1, 1.5, 2\}$

```
> lambda <- c(1, 1.5, 2)
> L = length(lambda)
> x <- seq(-5, 5, length = 501)
> for (i in 1:L) {
    Density = dsnorm(x, mean = 0, sd = 1, lambda = lambda[i])
    if (i == 1)
        plot(x, Density, type = "l", col = i, ylim = c(0, 0.5))
    else lines(x, Density, col = i)
 }
> title(main = "Density")
> grid()
> for (i in 1:L) {
    Probability = psnorm(x, mean = 0, sd = 1, lambda = lambda[i])
    if (i == 1)
        plot(x, Probability, type = "l", col = i)
    else lines(x, Probability, col = i)
    grid()
 }
> title(main = "Probability")
> grid()
> lambda <- 1/lambda
> for (i in 1:L) {
    Density = dsnorm(x, mean = 0, sd = 1, lambda = lambda[i])
    if (i == 1)
        plot(x, Density, type = "l", col = i, ylim = c(0, 0.5))
    else lines(x, Density, col = i)
    grid()
 }
> title(main = "Density")
> grid()
> for (i in 1:length(lambda)) {
    Probability = psnorm(x, mean = 0, sd = 1, lambda = lambda[i])
    if (i == 1)
        plot(x, Probability, type = "l", col = i)
```

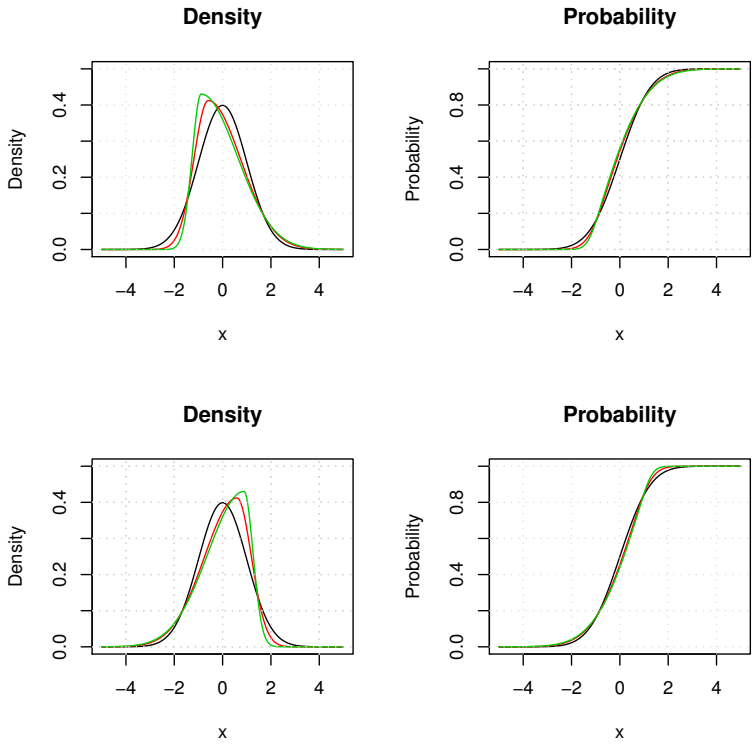

FIGURE 22.1: Skewed Nornal Distribution

```
    else lines(x, Probability, col = i)
 }
> title(main = "Probability")
> grid()
```

22.4 EXERCISE

1. Write a function `qsnorm()` which computes the quantile function of the skew normal distribution.

2. Write a function `snormFit()` which estimates the distributional parameters using the maximum log-likelihood approach.

CHAPTER 23

JARQUE-BERA HYPOTHESIS TEST

23.1 ASSIGNMENT

Write a R function for the Jarque-Bera Test to test the hypothesis if a series of financial returns is normally distributed or not.

The JarqueÃśBera test is a goodness-of-fit measure of departure from normality, based on the sample kurtosis and skewness, Jarque and Bera, 1980. The test statistic *JB* is defined as

$$JB = \frac{n}{6}\left(S^2 + \frac{1}{4}K^2\right)$$

where n is the number of observations, S is the sample skewness

$$S = \frac{\hat{\mu}_3}{\hat{\sigma}^3} = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^3}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2\right)^{3/2}}$$

and K is the sample kurtosis:

$$K = \frac{\hat{\mu}_4}{\hat{\sigma}^4} - 3 = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^4}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2\right)^2} - 3$$

Here $\hat{\mu}_3$ and $\hat{\mu}_4$ are the estimates of third and fourth central moments, respectively, $\bar{x}$ is the sample mean, and $\hat{\sigma}^2$ is the estimate of the second central moment, the variance. The statistic *JB* has an asymptotic chi-square distribution with two degrees of freedom and can be used to test the null hypothesis that the data are from a normal distribution. The null hypothesis is a joint hypothesis of the skewness being zero and the excess kurtosis being 0. As the definition of *JB* shows, any deviation from this increases the JB statistic, Wikipedia 2009.

References

Carlos M. Jarque and Anil K. Bera, 1980,
Efficient Tests for Normality, Homoskedasticity and Serial Independence of Regression Residuals,
Economics Letters 6, 255-259

Wikipedia, Jarque-Bera Test, 2010,
http://en.wikipedia.org/wiki/Jarque-Bera_test

23.2 R IMPLEMENTATION

The following R function performs the Jarque-Bera test, to test if a vector of financial returns is normally distributed. The function takes as argument x the vector of returns, and returns an object of class `"htest"`.

```
> jarque.bera.test <- function (x)
 {
     # Borrowed from the contributed R package tseries
     # Author: Adrian Trapletti

     # Assign a Name to the Data Vector:
     DNAME <- deparse(substitute(x))

     # Compute Statistics:
     n <- length(x)
     m1 <- sum(x)/n
     m2 <- sum((x - m1)^2)/n
     m3 <- sum((x - m1)^3)/n
     m4 <- sum((x - m1)^4)/n
     b1 <- (m3/m2^(3/2))^2
     b2 <- (m4/m2^2)
     STATISTIC <- n * b1/6 + n * (b2 - 3)^2/24
     names(STATISTIC) <- "X-squared"

     # Set the Number of Degrees of Freedom:
     PARAMETER <- 2
     names(PARAMETER) <- "df"

     # Compute the p-Value:
     PVAL <- 1 - pchisq(STATISTIC, df = 2)

     # Return Value:
     structure(list(
         statistic = STATISTIC,
         parameter = PARAMETER,
         p.value = PVAL,
         method = "Jarque Bera Test",
         data.name = DNAME),
         class = "htest")
 }
```

23.3 EXAMPLES

We want to test Swiss market indices if they are normal distributed. These are the Swiss Bond Index, SBI, the Swiss REIT Index, SII, and the Swiss Performance Index, SPI.
The data set is downloadable from the r-forge web site as a semicolon separated csv file. The file has 7 columns, the first holds the dates, the next 3 the SBI, SII, and SPI indices, and the last three Swiss pension fund index benchmarks.
Download the data and select columns 2 to 4

```
> library(fEcofin)
> data(SWXLP)
> x <- SWXLP[, 2:4]
> head(x)
    SBI    SPI    SII
1 95.88 5022.9 146.26
2 95.68 4853.1 146.27
3 95.67 4802.8 145.54
4 95.54 4861.4 146.10
5 95.58 4971.8 146.01
6 95.58 4982.3 146.36
```

Compute daily percentual returns

```
> x = diff(log(100 * as.matrix(x)))
```

and compute the tests statistic and p-values

```
> jarque.bera.test(x[, "SBI"])
Jarque Bera Test

data:  x[, "SBI"]
X-squared = 216.23, df = 2, p-value < 2.2e-16
> jarque.bera.test(x[, "SII"])
Jarque Bera Test

data:  x[, "SII"]
X-squared = 541.07, df = 2, p-value < 2.2e-16
> jarque.bera.test(x[, "SPI"])
Jarque Bera Test

data:  x[, "SPI"]
X-squared = 2192.7, df = 2, p-value < 2.2e-16
```

The X-squared statistic shows large increasing values for the three indices with a vanishing p-value. So the SBI, SII, and SPI show strong deviations from normality with increasing strength, as we would expect. All three series are rejected to be normal distributed. Let us have a look on the quantile-quantile plot which confirms these results.

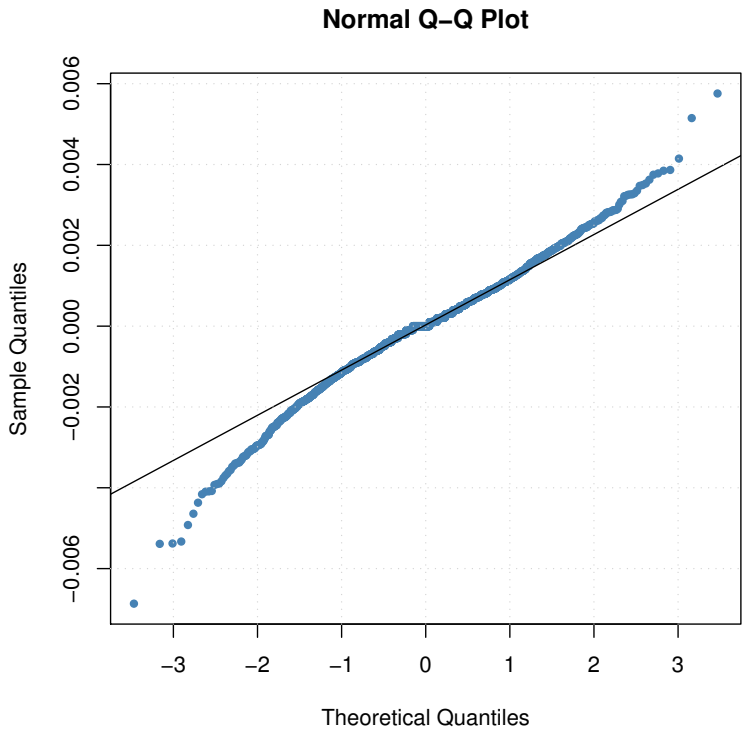

FIGURE 23.1: Quantile-Quantile Plots for the SBI, SII, and SPI

```
> qqPlot = function(x) {
     qqnorm(x, pch = 19, col = "steelblue", cex = 0.7)
     qqline(x)
     grid()
 }
> qqPlot(x[, "SBI"])
> qqPlot(x[, "SII"])
> qqPlot(x[, "SPI"])
```

CHAPTER 24

PCA ORDERING OF ASSETS

24.1 ASSIGNMENT

In this case study we will use the Principal Component Analysis, PCA, to order the individual instruments in a set of financial assets.

References

Joe H. Ward, 1963,
Hierarchical Grouping to Optimize an Objective Function,
Journal of the American Statistical Ass. 58, 236–244

Trevor Hastie, Robert Tibshirani, and Jerome Friedman, 2009,
The Elements of Statistical Learning,
Chapter: 14.3.12 Hierarchical clustering,
ISBN 0-387-84857-6, New York, Springer, 520Ãś-528,
http://www-stat.stanford.edu/ hastie/local.ftp/Springer/ESLII_print3.pdf

Wikipedia, Hierarchical Clustering, 2010,
http://en.wikipedia.org/wiki/Hierarchical_clustering

24.2 R IMPLEMENTATION

We proceed as follows: 1 transform the input in a numeric data matrix where the columns are the instruments, and the rows are the records in time, 2 compute the correlation matrix, 3 compute eigenvectors and eigenvalues of the correlation matrix, 4 and finally order the instruments

```
> arrangeAssets <- function(x, ...)
 {
     # Arguments:
     #   x - the set of assets, usually a multivariate time series
```

```
    # 1 Transform x Into a Matrix Object:
    x = as.matrix(x)

    # 2 Compute Correlation Matrix:
    x.cor = cor(x, ...)

    # 3 Compute Einvectors and Eigenvalues:
    x.eigen = eigen(x.cor)$vectors[, 1:2]
    e1 = x.eigen[, 1]
    e2 = x.eigen[, 2]

    # 4 Finally Order the Assets:
    Order = order(ifelse(e1 > 0, atan(e2/e1), atan(e2/e1)+pi))
    ans = colnames(as.matrix(x))[Order]

    # Return Value
    ans
}
```

To display the similarities graphically we write the following plot function for the ratio of eigenvalues

```
> similarityPlot <- function(x, ...)
 {
    # Order Assets:
    x.cor = cor(as.matrix(x), ...)
    x.eig = eigen(x.cor)$vectors[, 1:2]
    e1 = x.eig[, 1]
    e2 = x.eig[, 2]

    # Plot Ordered Assets:
    plot(e1, e2, col = 'white', ann = FALSE,
        xlim = range(e1, e2), ylim = range(e1, e2))
    abline(h = 0, lty = 3, col = "grey")
    abline(v = 0, lty = 3, col = "grey")
    arrows(0, 0, e1, e2, cex = 0.5, col = "steelblue", length = 0.1)
    text(e1, e2, rownames(x.cor))
    title(main = "Eigenvalue Ratio Plot", sub = "",
        xlab = "Eigenvalue 1", ylab = "Eigenvalue 2")

    # Return Value:
    invisible()
 }
```

24.3 EXAMPLES

Load the Swiss Pension Fund benchmark data set available from the (fBasics) package.

```
> require(fBasics)
> assets = 100 * LPP2005REC[, 1:6]
> head(round(assets, 5), 20)
GMT
```

```
                SBI      SPI      SII      LMI      MPI      ALT
2005-11-01 -0.06127  0.84146 -0.31909 -0.11089  0.15481 -0.25730
2005-11-02 -0.27620  0.25193 -0.41176 -0.11759  0.03429 -0.11416
2005-11-03 -0.11531  1.27073 -0.52094 -0.09925  1.05030  0.50074
2005-11-04 -0.32358 -0.07028 -0.11272 -0.11985  1.16796  0.94827
2005-11-07  0.13110  0.62052 -0.17958  0.03604  0.27096  0.47240
2005-11-08  0.05393  0.03293  0.21034  0.23270  0.03468  0.08536
2005-11-09 -0.25450 -0.23782 -0.18980 -0.20396  0.16927  0.36029
2005-11-10  0.10034  0.09221  0.10264  0.14398 -0.01717  0.24225
2005-11-11  0.06170  1.33349  0.04615  0.06522  0.73486  1.07096
2005-11-14  0.06936 -0.46931 -0.08720 -0.06958  0.00101 -0.10023
2005-11-15  0.01541  0.12669 -0.60734  0.17797 -0.01486 -0.14348
2005-11-16  0.29997 -0.71875  0.02065  0.27734  0.38712  0.01916
2005-11-17 -0.13064  0.76581 -0.11362  0.06734  0.51708  0.55882
2005-11-18 -0.22326  1.25272 -0.30534 -0.21044  0.56770  0.54052
2005-11-21  0.11554  0.26597  0.06218  0.23494  0.16780  0.21094
2005-11-22 -0.02310  0.21425 -0.17628  0.08838  0.34802  0.52915
2005-11-23  0.06928  0.35671  0.11410 -0.03154  0.18702  0.34939
2005-11-24  0.20754 -0.25595 -0.01037  0.17074  0.05054 -0.16492
2005-11-25 -0.06145  0.33748  0.17609  0.06820  0.25185  0.24512
2005-11-28 -0.03074 -0.98167 -0.21757 -0.00009 -0.81627 -0.55233
> end(assets)
GMT
[1] [2007-04-11]
```

The loaded data series is a (timeSeries) object with 377 daily records

```
> class(assets)
[1] "timeSeries"
attr(,"package")
[1] "timeSeries"
> dim(assets)
[1] 377   6
```

Here we have extracted the instruments from column 1 to 6. The series have been multiplied by 100 to observe percentaged returns. Let us order the instruments

```
> names(assets)
[1] "SBI" "SPI" "SII" "LMI" "MPI" "ALT"
```

yielding

```
> arrangeAssets(assets)
[1] "LMI" "SBI" "SII" "SPI" "MPI" "ALT"
```

Then display the similarities of the instruments in an eigenvalue-ratio plot

```
> similarityPlot(assets)
```

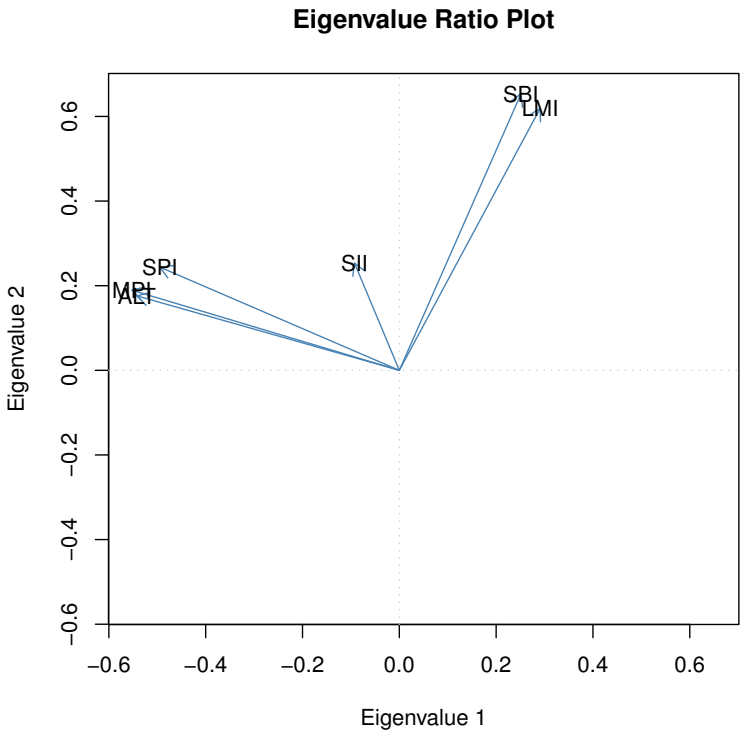

FIGURE 24.1: Similarity Plot of Swiss Pension Fund Benchmark

CHAPTER 25

CLUSTERING OF ASSET RETURNS

25.1 ASSIGNMENT

In statistics, hierarchical clustering is a method of cluster analysis that computes a hierarchy of clusters. We can use this approach to identify groups of assets in a portfolio and to arrange them in a hierarchical way. These hierarchical clusters are usually presented in a dendrogram, which is a tree-like structure.
Use the R's base function `hclust()` to investigate the hierarchical grouping of the assets and benchmarks from the Swiss pension fund series.

References

Joe H. Ward, 1963
Hierarchical Grouping to Optimize an Objective Function,
Journal of the American Statistical Association58, 236–244

Trevor Hastie, Robert Tibshirani, and Jerome Friedman,
The Elements of Statistical Learning,
Chapter: 14.3.12 Hierarchical clustering,
ISBN 0-387-84857-6, New York, Springer, 2009, 520-528,
http://www-stat.stanford.edu/ hastie/local.ftp/Springer/ESLII_print3.pdf

Wikipedia, Hierarchical Clustering, 2010,
http://en.wikipedia.org/wiki/Hierarchical_clustering

25.2 R IMPLEMENTATION

Let us write a function `clusteredAssets()` which takes a multivariate time series or a matrix of financial returns as input, and clusters them

hierarchically. We use R's base functions `t()`, `dist()` and `hclust()` to perform this task.
Given a matrix or data frame `x`, then the `t()`

```
> args(t)

function (x)
NULL
```

returns the transpose of `x`.
The function `dist()`

```
> args(dist)

function (x, method = "euclidean", diag = FALSE, upper = FALSE,
    p = 2)
NULL
```

computes and returns the distance matrix computed by using the specified distance measure to compute the distances between the rows of a data matrix. The distance measure to be used must be one of "euclidean", "maximum", "manhattan", "canberra", "binary" or "minkowski". `dist()` returns an object of class "dist".
The function `hclust()`

```
> args(hclust)

function (d, method = "complete", members = NULL)
NULL
```

does a hierarchical cluster analysis on a set of dissimilarities and methods for analyzing it. The help page states: "Initially, each object is assigned to its own cluster and then the algorithm proceeds iteratively, at each stage joining the two most similar clusters, continuing until there is just a single cluster. At each stage distances between clusters are recomputed by the Lance-Williams dissimilarity update formula according to the particular clustering method being used." The agglomeration method to be used is one of "ward", "single", "complete", "average", "mcquitty", "median" or "centroid". The function returns an object of class `"hclust"` for which a plot method is available

```
> clusteredAssets <- function(x, dist = "euclidean", method = "complete") {
     x = as.matrix(x)
     dist = dist(t(x), method = dist)
     clustering = hclust(dist, method = method)
     clustering
 }
```

Like the function `hclust()` the function `cluisteredAssets()` returns an object of class `"hclust"`.

25.3 EXAMPLES

In this example we group hierarchically the 6 asset classes and the three benchmark series from the Swiss pension fund benchmark series.
The data set is downloadable from the r-forge web site as a semicolon separated csv file. The file has 10 columns, the first holds the dates, the next 6 the assets, and the last three the benchmarks.

```
> require(fEcofin)
> require(timeSeries)
> x = as.timeSeries(LPP2005REC)
> names(x)
[1] "SBI"   "SPI"   "SII"   "LMI"   "MPI"   "ALT"   "LPP25" "LPP40" "LPP60"
```

Then we group the return series using the default settings: the `"euclidean"` distance measure, and the `"complete"` linkage method.
Show the distance matrix obtained from the financial return series

```
> dist(t(x))
           SBI      SPI      SII      LMI      MPI      ALT    LPP25    LPP40
SPI   0.154286
SII   0.060705 0.151784
LMI   0.020588 0.154386 0.060881
MPI   0.148463 0.105689 0.146625 0.150137
ALT   0.118405 0.108128 0.119690 0.119379 0.069716
LPP25 0.038325 0.122376 0.058179 0.040201 0.112964 0.084995
LPP40 0.059894 0.107120 0.070894 0.061389 0.092023 0.065881 0.022207
LPP60 0.088714 0.089892 0.092814 0.089874 0.066113 0.046772 0.051229 0.029056
```

and then compute the clusters

```
> clusters = clusteredAssets(x)
> clusters
Call:
hclust(d = dist, method = method)

Cluster method   : complete
Distance         : euclidean
Number of objects: 9
```

The result of the hierarchical clustering is shown in the dendrogram plot.

```
> plot(clusters)
```

25.4 EXERCISES

Compare the dendrogram as returned from alternative distance measures and linkage methods with the one obtained with the default settings, `dist="euclidean"` and `method="complete"`.

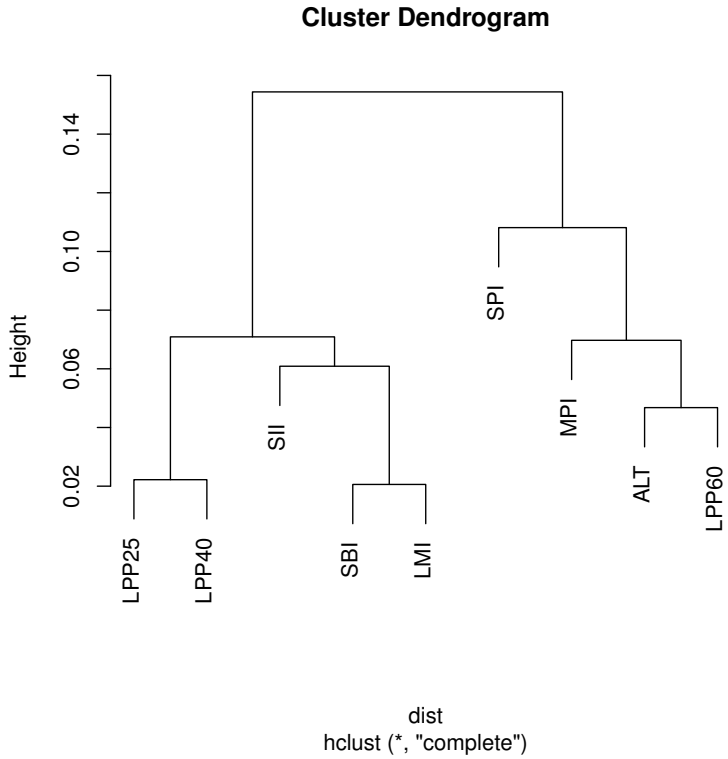

FIGURE 25.1: LPP Dendrogram Plot.

PART VII

CASE STUDIES: OPTION VALUATION

CHAPTER 26

BLACK SCHOLES OPTION PRICE

26.1 ASSIGNMENT

Black and Scholes succeeded in solving their differential equation to obtain exact formulas for the prices of European call and put options. The expected value of an European call option at maturity in a risk neutral world is

$$E[max(0, S_T - X)]$$

where E denotes the expected value, S_T the price of the underlying at maturity, and X the strike price.

The Black and Scholes formula

The price of a European call option c at time t is the discounted value at the risk free rate of interest r, that is,

$$c = e^{-r(T-t)} E[max(0, S_T - X)]$$

$\ln S_T$ has the probability distribution

$$\ln S_T - \ln S \sim \mathcal{N}\left((u - \frac{1}{2}\sigma^2)(T-t), \sigma(T-t)^{1/2}\right)$$

Evaluating the expectation value $E[max(0, S_T - X)]$ is an application of integral calculus, yielding

$$\begin{aligned} c &= S\mathcal{N}(d_1) - Xe^{-r(T-t)}\mathcal{N}(d_2) \\ d_1 &= \frac{\ln S/X + (r+\sigma^2/2)(T-t)}{\sigma(T-t)^{1/2}} \\ d_2 &= \frac{\ln S/X + (r-\sigma^2/2)(T-t)}{\sigma(T-t)^{1/2}} = d_1 - \sigma(T-t)^{1/2} \end{aligned} \tag{26.1}$$

and $\mathcal{N}$ is the cumulative distribution function for a standardized normal variable. The value of an European put can be calculated in a similar way, the result is

$$p = Xe^{-r(T-t)}N(-d_2) - SN(-d_1)\,.$$

The formula can be used as the starting point to price several kinds of options including European options on a stock with cash dividends, options on stock indexes, options on futures, and currency options.

The generalized Black and Scholes formula

The general version of the Black-Scholes model incorporates the cost-of-carry term b. It can be used to price European options on stocks, stocks paying a continuous dividend yield, options on futures, and currency options.

$$\begin{aligned} c_{GBS} &= Se^{(b-r)T}N(d_1) - Xe^{-rT}N(d_2)\,, \\ p_{GBS} &= Xe^{-rT}N(-d_2) - Se^{(b-r)T}N(-d_1)\,, \end{aligned} \tag{26.2}$$

where

$$\begin{aligned} d_1 &= \frac{\ln(S/X)+(b+\sigma^2/2)T}{\sigma\sqrt{T}}\,, \\ d_2 &= \frac{\ln(S/X)+(b-\sigma^2/2)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}\,. \end{aligned} \tag{26.3}$$

and b is the cost-of-carry rate of holding the underlying security.

- $b = r$ gives the Black-Scholes (1972) stock option model,
- $b = r - q$ gives the Mertom (1973) stock option model with continuous dividend yield q,
- $b = 0$ gives the Black (1976) futures option model, and
- $b = r - r_f$ gives the Garman and Kohlhagen (1983) currency option model.

References

Black Fischer and Myron Scholes, 1973,
The Pricing of Options and Corporate Liabilities,
Journal of Political Economy 81, 637–654

Robert C. Merton, 1973,
Theory of Rational Option Pricing,

Bell Journal of Economics and Management Science 1, 141–183
http://jstor.org/stable/3003143

John C. Hull, 1997,
Options, Futures, and Other Derivatives, Prentice Hall

Wikipedia, BlackÃśScholes, 2010,
http://en.wikipedia.org/wiki/Black%E2%80%93Scholes

26.2 R IMPLEMENTATION

Let us write a function `BlackScholes()` to compute the call and put price of the Black Scholes option. The arguments for the function are the call/put `TypeFlag`, the price of the underlying S, the strike price `X`, the time to maturity `Time`, the interest rate `r`, the cost of carry term `b`, and the `volatility`.

```
> BlackScholes <- function(TypeFlag=c("c", "p"), S, X, Time, r, b, sigma)
 {
     # Check Type Flag:
     TypeFlag = TypeFlag[1]

     # Compute d1 and d2:
     d1 = ( log(S/X) + (b+sigma*sigma/2)*Time ) / (sigma*sqrt(Time))
     d2 = d1 - sigma*sqrt(Time)

     # Compute Option Price:
     if (TypeFlag == "c")
         price = S*exp((b-r)*Time)*pnorm(d1) - X*exp(-r*Time)*pnorm(d2)
     else if (TypeFlag == "p")
         price = X*exp(-r*Time)*pnorm(-d2) - S*exp((b-r)*Time)*pnorm(-d1)

     # Save Parameters:
     param <- list(TypeFlag=TypeFlag, S=S, X=X, Time=Time, r=r, b=b, sigma=sigma)
     ans <- list(parameters=param, price=price, option = "Black Scholes")
     class(ans) <- c("option", "list")

     # Return Value:
     ans
 }
```

The function returns a list with three entries, the `$parameters`, the `$price`, and the name of the `$option`.
To return the result returned from the list object in a nicely printed form we write a S3 print method.

```
> print.option <- function(x, ...)
 {
     # Parameters:
     cat("\nOption:\n ")
     cat(x$option, "\n\n")
```

```
    # Parameters:
    cat("Parameters:\n")
    Parameter = x$parameters
    Names = names(Parameter)
    Parameter = cbind(as.character(Parameter))
    rownames(Parameter) = Names
    colnames(Parameter) = "Value:"
    print(Parameter, quote = FALSE)

    # Price:
    cat("\nOption Price:\n")
    cat(x$price, "\n")

    # Return Value:
    invisible()
}
```

26.3 EXAMPLES

European options on a stock with cash dividends

Consider an European call option on a stock that will pay out a dividend of two, three and six months from now. The current stock price is 100, the strike is 90, the time to maturity on the option is 9 months, the risk free rate is 10% and the volatility is 25%. First calculate the stock price minus the present value of the value of the cash dividends and then use the Black-Scholes formula to calculate the call price. The result will be 15.6465, as returned from the `@price` slot.

```
> S = 100 - 2 * exp(-0.1 * 0.25) - 2 * exp(-0.1 * 0.5)
> r = 0.1
> BlackScholes("c", S = S, X = 90, Time = 0.75, r = r, b = r, sigma = 0.25)

Option:
 Black Scholes

Parameters:
         Value:
TypeFlag c
S        96.146921326942
X        90
Time     0.75
r        0.1
b        0.1
sigma    0.25

Option Price:
15.647
```

Options on stock indexes

Consider an European put option with 6 months to expiry. The stock index is 100, the strike price is 95, the risk-free interest rate is 10%, the dividend yield is 5% per annum, and the volatility is 25%. The result for the put price will be 2.4648:

```
> r = 0.1
> q = 0.05
> BlackScholes("p", S = 100, X = 95, Time = 0.5, r = r, b = r -
     q, sigma = 0.2)
Option:
 Black Scholes

Parameters:
         Value:
TypeFlag p
S        100
X        95
Time     0.5
r        0.1
b        0.05
sigma    0.2

Option Price:
2.4648
```

Options on futures

Consider an European Option on the brent blend futures with nine months to expiry. The futures price USD 19, the risk-free interest rate is 10%, and the volatility is 28%. The result for the call price will be 1.7011 and the price for the put will be the same:

```
> FuturesPrice = 19
> b = 0
> BlackScholes("c", S = FuturesPrice, X = 19, Time = 0.75, r = 0.1,
     b = b, sigma = 0.28)
Option:
 Black Scholes

Parameters:
         Value:
TypeFlag c
S        19
X        19
Time     0.75
r        0.1
b        0
sigma    0.28

Option Price:
```

```
1.7011
```

Currency options

Consider an European call USD put DEM option with six months to expiry. The USD/DEM exchange rate is 1.5600, the strike price is 1.6000, the domestic risk-free interest rate in Germany is 6%, the foreign risk-free interest rate in the United States is 8% per annum, and the volatility is 12%. The result for the call price will be 0.0291:

```
> r = 0.06
> rf = 0.08
> BlackScholes("c", S = 1.56, X = 1.6, Time = 0.5, r = r, b = r -
     rf, sigma = 0.12)
Option:
 Black Scholes

Parameters:
         Value:
TypeFlag c
S        1.56
X        1.6
Time     0.5
r        0.06
b        -0.02
sigma    0.12

Option Price:
0.029099
```

CHAPTER 27

BLACK SCHOLES OPTION GREEKS

27.1 ASSIGNMENT

Recall from the Black-Scholes formula in the previous case study that the price of an option depends upon just five variables

- the current asset price,
- the strike price,
- the time to maturity,
- the volatility, and
- the interest rate.

One of these, the strike price, is normally fixed in advance and therefore does not change. That leaves the remaining four variables. We can now define four quantities, each of which measures how the value of an option will change when one of the input variables changes while the others remain the same.

Delta

Delta means the sensitivity of the option price to the movement in the underlying asset.

$$\Delta_{call} = \frac{\partial c}{\partial S} = e^{(b-r)T} N(d_1) > 0$$

$$\Delta_{put} = \frac{\partial p}{\partial S} = e^{(b-r)T} [N(d_1) - 1] < 0$$

Theta

Theta is the options sensitivity to small change in time to maturity. As time to maturity decreases, it is normal to express the Theta as minus the partial derivative with respect to time.

$$\Theta_{call} = \frac{\partial c}{\partial T} = -\frac{Se^{(b-r)T} n(d_1)\sigma}{2\sqrt{T}} - (b-r)Se^{(b-r)T} N(d_1) - rXe^{-rT} N(d_2)$$

$$\Theta_{put} = \frac{\partial p}{\partial T} = -\frac{Se^{(b-r)T} n(d_1)\sigma}{2\sqrt{T}} + (b-r)Se^{(b-r)T} N(-d_1) - rXe^{-rT} N(-d_2)$$

Vega

The Vega is the option's sensitivity to a small movement in the volatility of the underlying asset. Note that that Vega is equal for call and put options

$$Vega_{call,put} = \frac{\partial c}{\partial \sigma} = \frac{\partial p}{\partial \sigma} = Se^{(b-r)T} n(d_1)\sqrt{T} > 0$$

Rho

The Rho is the options sensitivity to a small change in the risk-free interest rate. For the call we have

$$\rho_{call} = \frac{\partial c}{\partial r} = TXe^{-rT} N(d_2) > 0 \ \ if \ \ b \neq 0\,,$$

$$\rho_{call} = \frac{\partial c}{\partial r} = -Tc < 0 \ \ if \ \ b = 0\,,$$

and for the put we have

$$\rho_{put} = \frac{\partial c}{\partial r} = -TXe^{-rT} N(-d_2) < 0 \ \ if \ \ b \neq 0\,,$$

$$\rho_{put} = \frac{\partial c}{\partial r} = -Tp < 0 \ \ if \ \ b = 0\,.$$

All four sensitivity measures so far have one thing in common: they all express how much an option's value will change for a unit change in one of the pricing variables. Since they measure changes in premium, Delta, Theta, Vega, and Rho will all be expressed in the same units as the option premium.

References

John C. Hull, 1997,
Options, Futures, and Other Derivatives, Prentice Hall

Wikipedia, BlackÃśScholes, 2010,
http://en.wikipedia.org/wiki/Black_Scholes

27.2 R IMPLEMENTATION

Write functions to compute the Greeks for of the Black and Scholes option. The function for the call and put price was calculated in the previous example:

```
> BlackScholes <- function(TypeFlag = c("c", "p"), S, X, Time,
      r, b, sigma) {
      TypeFlag = TypeFlag[1]
      d1 = (log(S/X) + (b + sigma * sigma/2) * Time)/(sigma * sqrt(Time))
      d2 = d1 - sigma * sqrt(Time)
      if (TypeFlag == "c")
          price = S * exp((b - r) * Time) * pnorm(d1) - X * exp(-r *
              Time) * pnorm(d2)
      else if (TypeFlag == "p")
          price = X * exp(-r * Time) * pnorm(-d2) - S * exp((b -
              r) * Time) * pnorm(-d1)
      param <- list(TypeFlag = TypeFlag, S = S, X = X, Time = Time,
          r = r, b = b, sigma = sigma)
      ans <- list(parameters = param, price = price, option = "Black Scholes")
      class(ans) <- c("option", "list")
      ans
  }
```

Let us start to implement the function `delta()` from the formula given above

Delta

```
> delta <- function(TypeFlag, S, X, Time, r, b, sigma) {
      d1 = (log(S/X) + (b + sigma * sigma/2) * Time)/(sigma * sqrt(Time))
      if (TypeFlag == "c")
          delta = exp((b - r) * Time) * pnorm(d1)
      else if (TypeFlag == "p")
          delta = exp((b - r) * Time) * (pnorm(d1) - 1)
      delta
  }
```

then the function for `theta()`

Theta

```
> theta <- function(TypeFlag, S, X, Time, r, b, sigma) {
      d1 = (log(S/X) + (b + sigma * sigma/2) * Time)/(sigma * sqrt(Time))
      d2 = d1 - sigma * sqrt(Time)
      NDF <- function(x) exp(-x * x/2)/sqrt(8 * atan(1))
      Theta1 = -(S * exp((b - r) * Time) * NDF(d1) * sigma)/(2 *
          sqrt(Time))
      if (TypeFlag == "c")
          theta = Theta1 - (b - r) * S * exp((b - r) * Time) *
              pnorm(+d1) - r * X * exp(-r * Time) * pnorm(+d2)
      else if (TypeFlag == "p")
          theta = Theta1 + (b - r) * S * exp((b - r) * Time) *
```

```
            pnorm(-d1) + r * X * exp(-r * Time) * pnorm(-d2)
    theta
}
```

then the function for `vega()`

Vega

```
> vega <- function(TypeFlag, S, X, Time, r, b, sigma) {
    NDF <- function(x) exp(-x * x/2)/sqrt(8 * atan(1))
    d1 = (log(S/X) + (b + sigma * sigma/2) * Time)/(sigma * sqrt(Time))
    vega = S * exp((b - r) * Time) * NDF(d1) * sqrt(Time)
    vega
}
```

and finally we implement the function for `rho()`

Rho

```
> rho <- function(TypeFlag, S, X, Time, r, b, sigma) {
    d1 = (log(S/X) + (b + sigma * sigma/2) * Time)/(sigma * sqrt(Time))
    d2 = d1 - sigma * sqrt(Time)
    CallPut = BlackScholes(TypeFlag, S, X, Time, r, b, sigma)$price
    if (TypeFlag == "c")
        if (b != 0)
            rho = Time * X * exp(-r * Time) * pnorm(d2)
        else rho = -Time * CallPut
    else if (TypeFlag == "p")
        if (b != 0)
            rho = -Time * X * exp(-r * Time) * pnorm(-d2)
        else rho = -Time * CallPut
    rho
}
```

27.3 EXAMPLES

Delta

Consider a futures option with six months to expiry. The futures price is 105, the strike price is 100, the risk-free interest rate is 10%, and the volatility is 36%. The Delta of the call price will be 0.5946 and the Delta of the put price −0.3566.

```
> delta("c", S = 105, X = 100, Time = 0.5, r = 0.1, b = 0, sigma = 0.36)
[1] 0.59463
> delta("p", S = 105, X = 100, Time = 0.5, r = 0.1, b = 0, sigma = 0.36)
[1] -0.3566
```

Theta

Consider an European put option on a stock index currently priced at 430. The strike price is 405, time to expiration is one month, the risk-free interest rate is 7% p.a., the dividend yield is 5% p.a., and the volatility is 20% p.a.. The Theta of the put option will be −31.1924.

```
> theta("p", S = 430, X = 405, Time = 1/12, r = 0.07, b = 0.07 -
     0.05, sigma = 0.2)
[1] -31.192
```

Vega

Consider a stock option with nine months to expiry. The stock price is 55, the strike price is 60, the risk-free interest rate is 10% p.a., and the volatility is 30% p.a.. What is the Vega? The result will be 18.9358.

```
> vega("c", S = 55, X = 60, Time = 0.75, r = 0.1, b = 0.1, sigma = 0.3)
[1] 18.936
```

Rho

Consider an European call option on a stock currently priced at 72. The strike price is 75, time to expiration is one year, the risk-free interest rate is 9% p.a., and the volatility is 19% p.a.. The result for Rho will be 38.7325.

```
> rho("c", S = 72, X = 75, Time = 1, r = 0.09, b = 0.09, sigma = 0.19)
[1] 38.733
```

CHAPTER 28

AMERICAN CALLS WITH DIVIDENDS

28.1 ASSIGNMENT

Roll (1977), Geske (19979) and Whaley (1982) have developed a formula for the valuation of an American call option on a stock paying a single dividend of D, with time to dividend payout t.

$$C = (S - De^{-rt})N(b_1) +$$

$$(S - Dw^{-rt})\, M(a_1, -b_1; -\sqrt{\frac{t}{T}}) - Xe^{-rt}$$

$$M(a_2, -b_2; -\sqrt{\frac{t}{T}}) - (X - D)e^{-rt}N(b_2)\,,$$

where

$$a_1 = \frac{\ln[(S - De^{-rt})/X] + (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$

$$a_2 = a_1 - \sigma\sqrt{T}$$

$$b_1 = \frac{\ln[(S - De^{-rt})/I] + (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$

$$b_2 = b_1 - \sigma\sqrt{T}$$

where $M(a, b; \rho)$ is the cumulative bivariate normal distribution function with upper integral limits a and b and correlation coefficient ρ. I is the critical ex-dividend stock price I that solves

$$c(I, X, T - t) = I + D - X\,,$$

where $c(I, X, T - t)$ is the value of the European call with stock price I and time to maturity $T - t$. If $D \leq X(1 - e^{-r(T-t)})$ or $I = \infty$, it will not be optimal

to exercise the option before expiration, and the price of the American option can be found by using the European Black and Scholes formula where the stock price is replaced with the stock price minus the present value of the dividend payment $S - De^{rt}$

References

R. Geske, 1979,
A Note on an Analytical Valuation Formula for Unprotected American Call Options on Stocks with Known Dividends,
Journal of Financial Economics 7, 375–380

R. Roll, 1977,
An Analytic Valuation Formula for Unprotected American Call Options on Stocks with Known Dividends,
Journal of Financial Economics 5, 251–258

R. E. Whaley, 1982,
Valuation of American Call Options on Dividend-Paying Stocks: Empirical Tests,
Journal of Financial Economics 10, 29–58

E.G. Haug, 1997,
The Complete Guide to Option Pricing Formulas,
McGraw-Hill, New York.

28.2 R IMPLEMENTATION

The function for the call and put price was calculated in the previous example:

```
> BlackScholes <- function(TypeFlag, S, X, Time, r, b, sigma)
 {
     # Compute d1 and d2:
     d1 = ( log(S/X) + (b+sigma*sigma/2)*Time ) / (sigma*sqrt(Time))
     d2 = d1 - sigma*sqrt(Time)

     # Compute Option Price:
     if (TypeFlag == "c")
         price = S*exp((b-r)*Time)*pnorm(d1) - X*exp(-r*Time)*pnorm(d2)
     else if (TypeFlag == "p")
         price = X*exp(-r*Time)*pnorm(-d2) - S*exp((b-r)*Time)*pnorm(-d1)

     # Return Value:
     price
 }
```

We also need an R function for the cumulative bivariate normal distribution. We implement the following approximation from Abramowitz Stegun and used in Haug (1997)

```
> CBND <- function (a, b, rho)
 {
    # Cumulative Bivariate Normal distribution:
    if (abs(rho) == 1) rho = rho - (1e-12) * sign(rho)
    X = c(0.24840615, 0.39233107, 0.21141819, 0.03324666, 0.00082485334)
    y = c(0.10024215, 0.48281397, 1.0609498, 1.7797294, 2.6697604)
    a1 = a/sqrt(2 * (1 - rho^2))
    b1 = b/sqrt(2 * (1 - rho^2))
    if (a <= 0 && b <= 0 && rho <= 0) {
        Sum1 = 0
        for (I in 1:5) {
            for (j in 1:5) {
                Sum1 = Sum1 + X[I] * X[j] * exp(a1 * (2 * y[I] -
                  a1) + b1 * (2 * y[j] - b1) + 2 * rho * (y[I] -
                  a1) * (y[j] - b1)) } }
        result = sqrt(1 - rho^2)/pi * Sum1
        return(result) }
    if (a <= 0 && b >= 0 && rho >= 0) {
        result = pnorm(a) - CBND(a, -b, -rho)
        return(result) }
    if (a >= 0 && b <= 0 && rho >= 0) {
        result = pnorm(b) - CBND(-a, b, -rho)
        return(result) }
    if (a >= 0 && b >= 0 && rho <= 0) {
        result = pnorm(a) + pnorm(b) - 1 + CBND(-a, -b, rho)
        return(result) }
    if (a * b * rho >= 0) {
        rho1 = (rho * a - b) * sign(a)/sqrt(a^2 - 2 * rho * a * b + b^2)
        rho2 = (rho * b - a) * sign(b)/sqrt(a^2 - 2 * rho * a * b + b^2)
        delta = (1 - sign(a) * sign(b))/4
        result = CBND(a, 0, rho1) + CBND(b, 0, rho2) - delta
        return(result) }
    invisible()
 }
```

Now we are ready to write an R function for the American Call approximation

```
> RollGeskeWhaley <- function(S, X, time1, Time2, r, D, sigma)
 {
    # Tolerance Settings:
    big = 1.0e+8
    eps = 1.0e-5

    # Compute Option Price:
    Sx = S - D * exp(-r * time1)
    if(D <= X * (1 - exp(-r*(Time2-time1)))) {
        result = BlackScholes("c", Sx, X, Time2, r, b=r, sigma)
        cat("\nWarning: Not optimal to exercise\n")
        return(result) }
    ci = BlackScholes("c", S, X, Time2-time1, r, b=r, sigma)
```

```
    HighS = S
    while ( ci-HighS-D+X > 0 && HighS < big ) {
        HighS = HighS * 2
        ci = BlackScholes("c", HighS, X, Time2-time1, r, b=r, sigma) }
    if(HighS > big) {
        result = BlackScholes("c", Sx, X, Time2, r, b=r, sigma)
        stop()}
    LowS = 0
    I = HighS * 0.5
    ci = BlackScholes("c", I, X, Time2-time1, r, b=r, sigma)

    # Search algorithm to find the critical stock price I
    while ( abs(ci-I-D+X) > eps && HighS - LowS > eps ) {
        if(ci-I-D+X < 0 ) HighS = I else LowS = I
        I = (HighS + LowS) / 2
        ci = BlackScholes("c", I, X, Time2-time1, r, b=r, sigma) }
    a1 = (log(Sx/X) + (r+sigma^2/2)*Time2) / (sigma*sqrt(Time2))
    a2 = a1 - sigma*sqrt(Time2)
    b1 = (log(Sx/I) + (r+sigma^2/2)*time1) / (sigma*sqrt(time1))
    b2 = b1 - sigma*sqrt(time1)
    result = Sx*pnorm(b1) + Sx*CBND(a1,-b1,-sqrt(time1/Time2)) -
        X*exp(-r*Time2)*CBND(a2,-b2,-sqrt(time1/Time2)) -
        (X-D)*exp(-r*time1)*pnorm(b2)

    # Return Value:
    result
}
```

28.3 EXAMPLES

Consider an American-style call option on a stock that will pay a dividend of 4 in exactly three months. The stock price is 80, the strike price is 82, time to maturity is four months, the risk-free interest rate is 6%, and the volatility is 30%.

```
> RollGeskeWhaley(S = 80, X = 82, time1 = 1/4, Time2 = 1/3, r = 0.06,
    D = 4, sigma = 0.3)
[1] 4.386
```

The result is 4.386, whereas the value of a similar European call would be 3.5107.

CHAPTER 29

MONTE CARLO OPTION PRICING

29.1 ASSIGNMENT

Let us write a simple Monte Carlo Simulator for estimating the price of a path dependent option. As an example and exercise we consider the Black and Scholes option to test the code and and the Asian Option as a path dependent option.
The simulator should be so general that we can allow for any random path and for any option payoff to be specified by the user through a function call.

References

P. Glasserman, 2004,
Monte Carlo Methods in Financial Engineering, Springer-Publishing New York, Inc.

29.2 R IMPLEMENTATION

Let us implement a function to estimate the price of an option by Monte Carlo simulation. We assume that we have already `R` functions to generate the price path `path.gen()` and to compute the payoff of the option `payoff.calc()`. These two function will become arguments of the simulator function. The Simulator requires as inputs the length of the time interval on the path, `delta.t`, the length of the path, `pathLength`, and the number of Monte Carlo steps, `mcSteps`, per Monte Carlo Loop, `mcLoops`. The total number of Monte Carlo steps is then given by `mcSteps*mcLoops`.
To make life easier we define the options parameters globally, these are for the Black and Scholes option: the `TypeFlag`, the price of the underlying `S`, the strike price `X`, the time to maturity `Time`, the interest rate `r`, the cost of carry term `b`, anbd the volatility `sigma`.

The simulator loops over the number of Monte Carlo loops, `mcLoops`, and simulates then in each loop `mcSteps` Monte Carlo steps. For this, first we generate the random innovations, second we calculate for each path the option price, and third we trace the simulation to display the result. The result will be stored and returned in the variable `iteration`.

```
> MonteCarloOption <- function(
    delta.t, pathLength, mcSteps, mcLoops, path.gen, payoff.calc)
 {
    # Arguments:
    #   delta.t     - The length of the time interval, by default one day
    #   pathLength  - Number of Time Intervals which add up to the path
    #   mcSteps     - The number of Monte Carlo Steps performed in one loop
    #   mcLoops     - The number of Monte Carlo Loops
    #   path.gen    - the generator for the MC paths
    #   payoff.calc - the payoff claculator function

    # Monte Carlo Simulation:
    delta.t <<- delta.t
    cat("\nMonte Carlo Simulation Path:\n\n")
    iteration = rep(0, length = mcLoops)

    cat("Loop:\t", "No\t")
    for ( i in 1:mcLoops ) {
        if ( i > 1) init = FALSE
        # 1 Generate Random Innovations:
          eps = matrix(rnorm(mcSteps*pathLength), nrow=mcSteps)
        # 2 Calculate for each path the option price:
          path = t(path.gen(eps))
          payoff = NULL
          for (j in 1:dim(path)[1])
              payoff = c(payoff, payoff.calc(path[, j]))
          iteration[i] = mean(payoff)
        # 3 Trace the Simualtion:
          cat("\nLoop:\t", i, "\t:", iteration[i], sum(iteration)/i )
    }
    cat("\n")

    # Return Value:
    iteration
 }
```

29.3 EXAMPLES

Now let us perform a Monte Carlo simulation. To test our simulator we run the Black and Scholes Call Option for which we know the exact result in the continuum lime where the length of the time interval `delta.t` vanishes.

Path generation

First we have to write an Rfunction which generates the option's price paths. For the Black and Scholes model this is just a Wiener path.

```
> wienerPath <- function(eps)
 {
     # Generate the Paths:
     path = (b-sigma*sigma/2)*delta.t + sigma*sqrt(delta.t)*eps

     # Return Value:
     path
 }
```

Payoff calculator

First we have to write an Rfunction which computes the option's payoff. For the Black and Scholes model this is payoff we have implemented in the previous examples into the `BlackScholeS()` function.

```
> plainVanillaPayoff <- function(path)
 {
     # Compute the Call/Put Payoff Value:
     ST = S*exp(sum(path))
     if (TypeFlag == "c") payoff = exp(-r*Time)*max(ST-X, 0)
     else if (TypeFlag == "p") payoff = exp(-r*Time)*max(0, X-ST)

     # Return Value:
     payoff
 }
```

Simulate

Now we are ready to estimate the option's price by Monte Carlo simulation. Define the parameters glovally

```
> TypeFlag <<- "c"
> S <<- 100
> X <<- 100
> Time <<- 1/12
> sigma <<- 0.4
> r <<- 0.1
> b <<- 0.1
```

and then start to simulate

```
> set.seed = 4711
> mc = MonteCarloOption(delta.t = 1/360, pathLength = 30, mcSteps = 5000,
     mcLoops = 20, path.gen = wienerPath, payoff.calc = plainVanillaPayoff)
Monte Carlo Simulation Path:

Loop:  No
```

```
Loop:  1  : 7.8775 7.8775
Loop:  2  : 3.9575 5.9175
Loop:  3  : 3.5745 5.1365
Loop:  4  : 5.4086 5.2045
Loop:  5  : 4.9063 5.1449
Loop:  6  : 5.0489 5.1289
Loop:  7  : 5.1139 5.1267
Loop:  8  : 5.3045 5.149
Loop:  9  : 2.9363 4.9031
Loop:  10  : 3.7013 4.7829
Loop:  11  : 5.537 4.8515
Loop:  12  : 6.8263 5.016
Loop:  13  : 4.1457 4.9491
Loop:  14  : 4.4251 4.9117
Loop:  15  : 3.402 4.811
Loop:  16  : 5.8018 4.873
Loop:  17  : 4.02 4.8228
Loop:  18  : 4.3781 4.7981
Loop:  19  : 6.146 4.869
Loop:  20  : 4.0902 4.8301
```

Note we have taken not too many Monte Carlo steps to finish the the simulation in a reasonable short execution time.

Plot the MC iteration path

```
> mcPrice = cumsum(mc)/(1:length(mc))
> plot(mcPrice, type = "l", main = "Arithmetic Asian Option", xlab = "Monte Carlo Loops",
     ylab = "Option Price")
> abline(h = 5.0118, col = "red")
> grid()
```

29.4 EXERCISES

Simulate an arithmetic Asian Option. The function to compute the payoff is

```
> arithmeticAsianPayoff <- function(path) {
     SM = mean(S * exp(cumsum(path)))
     if (TypeFlag == "c")
         payoff = exp(-r * Time) * max(SM - X, 0)
     else if (TypeFlag == "p")
         payoff = exp(-r * Time) * max(0, X - SM)
     payoff
 }
```

The rest remains unchanged.

PART VIII

CASE STUDIES: PORTFOLIO DESIGN

CHAPTER 30

MEAN-VARIANCE MARKOWITZ PORTFOLIO

30.1 ASSIGNMENT

Following Markowitz 1952 we define the problem of portfolio selection as follows:

$$\begin{aligned} \min_{w} \quad & w^T \hat{\Sigma}\, w \\ s.t. \quad & \\ & w^T \hat{\mu} = \overline{r} \\ & w^T 1 = 1 \end{aligned}$$

The formula expresses that we minimize the variance-covariance risk $\overline{\sigma}^2 = w^T\, \hat{\Sigma}\, w$, where the matrix $\hat{\Sigma}$ is an estimate of the covariance of the assets. The vector w denotes the individual investments subject to the condition $w^T 1 = 1$ that the available capital is fully invested. The expected or target return $\overline{r}$ is expressed by the condition $w^T \hat{\mu} = \overline{r}$, where the p-dimensional vector $\hat{\mu}$ estimates the expected mean of the assets.
The unlimited short selling portfolio can be solved analytically. However, if the weights are bounded by zero, which forbids short selling, then the optimization has to be done numerically. The structure of the portfolio problems is quadratic and thus we can use a quadratic solver to compute the weights of the portfolio. Then we consider as the standard Markowitz portfolio problem a portfolio which sets box and group constraints on the weights:

$$\begin{aligned} \min_{w} \quad & w^T \Sigma\, w \\ s.t. \quad & \\ & Aw \leq b \end{aligned}$$

It can be shown that, if Σ is a positive definite matrix, the Markowitz portfolio problem is a convex optimization problem. As such, its local optimal solutions are also global optimal solutions.
The contributed R package `quadprog` provides the function `solve.QP()`, which interfaces a FORTRAN subroutine. This subroutine implements the dual method of Goldfarb and Idnani, 1982 and 1983, for solving quadratic programming problems of the form $min(-c^T x + 1/2 x^T C x)$ with the constraints $A^T x \geq b$. We use in the following this solver for optimizing the long only constrained mean-variance Markowitz portfolio optimization problem.

References

Harry M. Markowitz, 1952, Portfolio Selection,
The Journal of Finance 7, 77–91

Wikipedia, Modern Portfolio Theory, 2010,
http://en.wikipedia.org/wiki/Modern_portfolio_theory

30.2 R IMPLEMENTATION

We take the quadratic solver `solve.QP()` from the contributed R package `quadprog`

```
> library(quadprog)
> args(solve.QP)
function (Dmat, dvec, Amat, bvec, meq = 0, factorized = FALSE)
NULL
```

This routine implements the dual method of Goldfarb and Idnani, 1982 and 1983, for solving quadratic programming problems of the form

$$min(-d^T b + 1/2 b^T D b)$$

with the constraints

$$A^T b \geq b_0.$$

The argument list of the solver has 7 elements:

```
Dmat        matrix appearing in the quadratic function to be minimized
dvec        vector appearing in the quadratic function to be minimized
Amat        matrix defining the constraints
bvec        vector holding the values of $b_0$ (defaults to zero).
meq         the first meq constraints are treated as equality constraints,
            all further as inequality constraints (defaults to 0)
```

the last element `factorized` we do not use here.
The function returns a list with the following components:

```
solution    vector containing the solution of the quadratic programming
            problem.
value       scalar, the value of the quadratic function at the solution
            unconstrained.solution vector containing the unconstrained
            minimizer of the quadratic function.
iterations  vector of length 2, the first component contains the number
            of iterations the algorithm needed, the second indicates how
            often constraints became inactive after becoming active first.
iact        vector with the indices of the active constraints at the
            solution.
```

Now we are ready to write a function to optimize the Markowitz portfolio for a set of asset returns and a given target return. The function body consists of two parts: 1 create the portfolio settings from the arguments, and 2 Optimize weights with the quadratic solver.

```
> portfolio <- function(assetReturns, targetReturn)
 {
    # Arguments:
    #   assetReturns - multivariate data set of asset returns
    #   target Return - the portfolios target return

    # 1 Create Portfolio Settings:
    nAssets = ncol(assetReturns)
    Dmat = cov(assetReturns)
    dvec = rep(0, times=nAssets)
    Amat = t(rbind(
        Return=colMeans(assetReturns),
        Budget=rep(1, nAssets),
        LongOnly=diag(nAssets)))
    bvec = c(
        Return=targetReturn,
        budget=1,
        LongOnly=rep(0, times=nAssets))
    meq = 2

    # 2 Optimize Weights:
    portfolio = solve.QP(Dmat, dvec, Amat, bvec, meq)
    weights = round(portfolio$solution, digits = 4)
    names(weights) = colnames(assetReturns)

    # Return Value:
    list(
        weights = 100*weights,
        risk = portfolio$value,
        return = targetReturn)
 }
```

30.3 EXAMPLES

As an example we consider the Swiss pension fund benchmark which we have compressed in 6 asset series and three benchmark series. The data

can be loaded from the Rmetrics package `fBasics`

```
> library(fBasics)
> assetReturns <- 100 * LPP2005REC[, 1:6]
> names(assetReturns)
[1] "SBI" "SPI" "SII" "LMI" "MPI" "ALT"
```

Note to get daily percentage returns we have multiplied the series with 100. For the target return we choose the value of the grand mean of the assets

```
> targetReturn <- mean(colMeans(assetReturns))
> targetReturn
[1] 0.043077
```

Then we optimize the portfolio

```
> portfolio <- portfolio(assetReturns, targetReturn)
> portfolio
$weights
  SBI   SPI   SII   LMI   MPI   ALT
 0.00  0.86 25.43 33.58  0.00 40.13

$risk
[1] 0.030034

$return
[1] 0.043077
```

Extract the weights from the returned list

```
> weights = portfolio$weights
> names(weights) = colnames(data)
> weights
[1]  0.00  0.86 25.43 33.58  0.00 40.13
```

Check if we are fully invested and the sum of the weighted returns yields the target return

```
> sum(weights)
[1] 100
> c(weightedReturn = round((weights %*% colMeans(assetReturns))[[1]],
     3), targetReturn = round(100 * targetReturn, 3))
weightedReturn   targetReturn
         4.308          4.308
```

Let us now create a pie chart for the assets with non zero weights

```
> args(pie)
function (x, labels = names(x), edges = 200, radius = 0.8, clockwise = FALSE,
    init.angle = if (clockwise) 90 else 0, density = NULL, angle = 45,
    col = NULL, border = NULL, lty = NULL, main = NULL, ...)
NULL
```

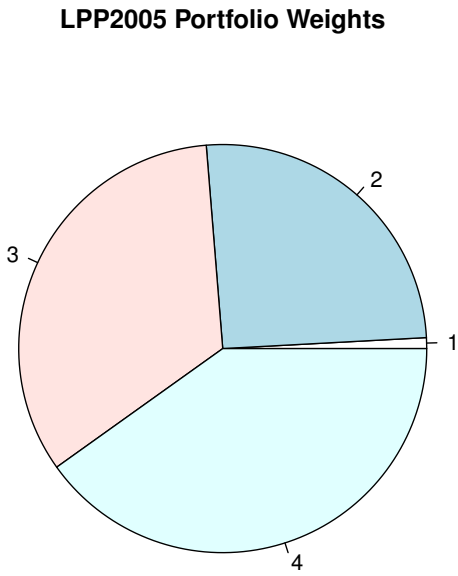

FIGURE 30.1: Pie Plot of Portfolio Weights

```
> Weights = weights[weights > 0]
```

```
> pie(Weights, labels = names(Weights))
> title(main = "LPP2005 Portfolio Weights")
```

CHAPTER 31

MARKOWITZ TANGENCY PORTFOLIO

31.1 ASSIGNMENT

Reward/risk profiles from the Markowitz portfolio of different combinations of a risky portfolio with a riskless asset, with expected return r_f, can be represented as a straight line in a risk versus reward plot, the so called capital market line, CML. The point where the CML touches the efficient frontier corresponds to the optimal risky portfolio. This portfolio is also called the mean–variance tangency portfolio. Mathematically, this can be expressed as the portfolio that maximizes the quantity

$$\begin{aligned} \max_{w} \quad & h(w) = \frac{\hat{\mu}^T w - r_f}{w^T \hat{\Sigma} w} \\ s.t. \quad & \\ & w^T \hat{\mu} = \overline{r} \\ & w^T 1 = 1 \end{aligned}$$

among all w. This quantity is precisely the *Sharpe ratio* introduced by Sharpe, 1994.
In the following we want to write an R function which computes for a mean–variance Markowitz portfolio, the tangency portfolio, i.e. the risk, the return, the weights and the Sharpe ratio in this point.

References

Harry M. Markowitz, 1952, Portfolio Selection,
The Journal of Finance 7, 77–91

Wikipedia, Modern Portfolio Theory, 2010,
http://en.wikipedia.org/wiki/Modern_portfolio_theory

Wikipedia, Sharpe Ratio, 2010,
http://en.wikipedia.org/wiki/Sharpe_ratio

31.2 R IMPLEMENTATION

We use a simplified version of the `portfolio()` function from the previous case study to compute the weights for mean-variance Markowitz portfolio with long only constraints.

```
> library(quadprog)
> portfolioWeights <- function(assetReturns, targetReturn)
 {
     nAssets = ncol(assetReturns)
     portfolio = solve.QP(
         Dmat = cov(assetReturns),
         dvec = rep(0, times=nAssets),
         Amat = t(rbind(Return=colMeans(assetReturns),
             Budget=rep(1, nAssets), LongOnly=diag(nAssets))),
         bvec = c(Return=targetReturn, budget=1,
             LongOnly=rep(0, times=nAssets)),
         meq=2)
     weights = portfolio$solution
     weights
 }
```

The tangency portfolio is then obtained by maximizing the Sharpe Ratio as a function of the target return. The steps are the following: 1 we write an internal function for Sharpe ratio, 2 then we optimize the weights for the tangency portfolio, and 3 we extract the characteristics of the tangency portfolio. These are the tangency portfolio's risk, its returns, the corresponding weights, and the resulting Sharpe ratio.
The function we use for optimization is the base `R` function `optim()`. This is a general-purpose optimizer based on NelderÃśMead, quasi-Newton and conjugate-gradient algorithms. We use the default settings. The optimizer allows for for box-constrained optimization, in our case the box is the range of the possible returns.
Note that to pass the weights and target risk value we have added attributes to the returns value of the `harpeRatio()` function

```
> tangencyPortfolio <-
 function (assetReturns, riskFreeRate=0)
 {
     # 1 Sharpe Ratio Function:
     sharpeRatio <- function(x, assetReturns, riskFreeRate)
     {
         targetReturn = x
         weights = portfolioWeights(assetReturns, targetReturn)
         targetRisk = sqrt( weights %*% cov(assetReturns) %*% weights )[[1]]
         ratio = (targetReturn - riskFreeRate)/targetRisk
         attr(ratio, "weights") <- weights
         attr(ratio, "targetRisk") <- targetRisk
```

```
        ratio
    }

    # 2 Optimize Tangency Portfolio:
    nAssets = ncol(assetReturns)
    mu = colMeans(assetReturns)
    Cov = cov(assetReturns)
    tgPortfolio <- optimize(
        f=sharpeRatio, interval=range(mu), maximum=TRUE,
        assetReturns=assetReturns, riskFreeRate=riskFreeRate)

    # 3 Tangency Portfolio Characteristics:
    tgReturn = tgPortfolio$maximum
    tgRisk = attr(tgPortfolio$objective, "targetRisk")
    tgWeights = attr(tgPortfolio$objective, "weights")
    sharpeRatio = sharpeRatio(tgReturn, assetReturns, riskFreeRate)[[1]]

    # Return Value:
    list(
        sharpeRatio=sharpeRatio,
        tgRisk=tgRisk, tgReturn=tgReturn, tgWeights=tgWeights)
}
```

31.3 EXAMPLES

As an example we consider as in the previous case study the Swiss pension fund benchmark. The data can be loaded from the Rmetrics package `fBasics`. We use daily percentage returns

```
> library(fBasics)
> assetReturns <- 100 * LPP2005REC[, 1:6]
> names(assetReturns)
[1] "SBI" "SPI" "SII" "LMI" "MPI" "ALT"
```

Now let us compute the tangency portfolio for a risk free rate.

```
> tangencyPortfolio(assetReturns, riskFreeRate = 0)
$sharpeRatio
[1] 0.18471

$tgRisk
[1] 0.15339

$tgReturn
[1] 0.028333

$tgWeights
[1] 0.0000e+00 4.8169e-04 1.8244e-01 5.7512e-01 4.7712e-18 2.4196e-01
```

Note depending on the assets, the tangency portfolio may not always exist, and thus our function may fail.

CHAPTER 32

LONG ONLY PORTFOLIO FRONTIER

32.1 ASSIGNMENT

The efficient frontier together with the minimum variance locus form the "upper border" and "lower border" lines of the set of all feasible portfolios. To the right the feasible set is determined by the envelope of all pairwise asset frontiers. The region outside of the feasible set is unachievable by holding risky assets alone. No portfolios can be constructed corresponding to the points in this region. Points below the frontier are suboptimal. Thus, a rational investor will hold a portfolio only on the frontier. Now we show how to compute the whole efficient frontier and the minimum variance locus of a mean-variance portfolio.

References

Harry M. Markowitz, 1952, Portfolio Selection,
The Journal of Finance 7, 77–91

Wikipedia, Modern Portfolio Theory, 2010,
http://en.wikipedia.org/wiki/Modern_portfolio_theory

32.2 R IMPLEMENTATION

We use a simplified version of the `portfolio()` function from the previous case study to compute the weights on the efficient frontier and the minimum variance locus.

```
> library(quadprog)
> portfolioWeights <- function(assetReturns, targetReturn)
 {
     nAssets = ncol(assetReturns)
     portfolio = solve.QP(
         Dmat = cov(assetReturns),
```

```
            dvec = rep(0, times=nAssets),
            Amat = t(rbind(Return=colMeans(assetReturns),
                Budget=rep(1, nAssets), LongOnly=diag(nAssets))),
            bvec = c(Return=targetReturn, budget=1,
                LongOnly=rep(0, times=nAssets)),
            meq=2)
        weights = portfolio$solution
        weights
    }
```

Now we write the function `portfolioFrontier()` which returns the weights of portfolios along the frontier with equidistant target returns. The number of efficient portfolios is given by `nPoints`

```
> portfolioFrontier <- function(assetReturns, nPoints=20)
 {
    # Number of Assets:
    nAssets = ncol(assetReturns)

    # Target Returns:
    mu = colMeans(assetReturns)
    targetReturns <- seq(min(mu), max(mu), length=nPoints)

    # Optimized Weights:
    weights = rep(0, nAssets)
    weights[which.min(mu)] = 1
    for (i in 2:(nPoints-1)) {
        newWeights = portfolioWeights(assetReturns, targetReturns[i])
        weights = rbind(weights, newWeights)
    }
    newWeights = rep(0, nAssets)
    newWeights[which.max(mu)] = 1
    weights = rbind(weights, newWeights)
    weights = round(weights, 4)
    colnames(weights) = colnames(assetReturns)
    rownames(weights) = 1:nPoints

    # Return Value:
    weights
}
```

Note the first and last portfolios are those from the assets with the highest and lowest reutns.

32.3 EXAMPLES

As an example we consider as in the previous case study the Swiss pension fund benchmark. The data can be loaded from the Rmetrics package `fBasics`. We use daily percentage returns

```
> library(fBasics)
> assetReturns <- 100 * LPP2005REC[, 1:6]
> names(assetReturns)
```

```
[1] "SBI" "SPI" "SII" "LMI" "MPI" "ALT"
```

Then we compute the weights for 20 portfolios on the efficient frontier and on the minimum variance locus

```
> weights = portfolioFrontier(assetReturns, nPoints = 20)
> print(weights)
      SBI    SPI    SII    LMI    MPI    ALT
1  1.0000 0.0000 0.0000 0.0000 0.0000 0.0000
2  0.5457 0.0000 0.0424 0.3887 0.0231 0.0000
3  0.3828 0.0000 0.0828 0.4783 0.0146 0.0415
4  0.2626 0.0000 0.1056 0.5377 0.0000 0.0941
5  0.1235 0.0000 0.1299 0.6113 0.0000 0.1352
6  0.0000 0.0000 0.1540 0.6685 0.0000 0.1775
7  0.0000 0.0000 0.1765 0.5950 0.0000 0.2286
8  0.0000 0.0023 0.1985 0.5217 0.0000 0.2775
9  0.0000 0.0048 0.2205 0.4484 0.0000 0.3263
10 0.0000 0.0073 0.2425 0.3752 0.0000 0.3750
11 0.0000 0.0098 0.2645 0.3020 0.0000 0.4238
12 0.0000 0.0123 0.2865 0.2287 0.0000 0.4725
13 0.0000 0.0148 0.3085 0.1555 0.0000 0.5213
14 0.0000 0.0173 0.3305 0.0822 0.0000 0.5700
15 0.0000 0.0198 0.3525 0.0090 0.0000 0.6188
16 0.0000 0.0263 0.2910 0.0000 0.0000 0.6827
17 0.0000 0.0333 0.2179 0.0000 0.0000 0.7488
18 0.0000 0.0403 0.1448 0.0000 0.0000 0.8149
19 0.0000 0.0473 0.0717 0.0000 0.0000 0.8809
20 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000
```

and save the target returns and target risks

```
> mu = colMeans(assetReturns)
> targetReturns = seq(min(mu), max(mu), length = nrow(weights))
> targetRisks = NULL
> for (i in 1:nrow(weights)) {
     newTargetRisk = sqrt(weights[i, ] %*% cov(assetReturns) %*%
         weights[i, ])
     targetRisks = c(targetRisks, newTargetRisk)
 }
```

Finally we plot the efficient frontier and the minimum variance locus

```
> plot(targetRisks, targetReturns, pch = 19)
> title(main = "LPP Bechmark Portfolio")
```

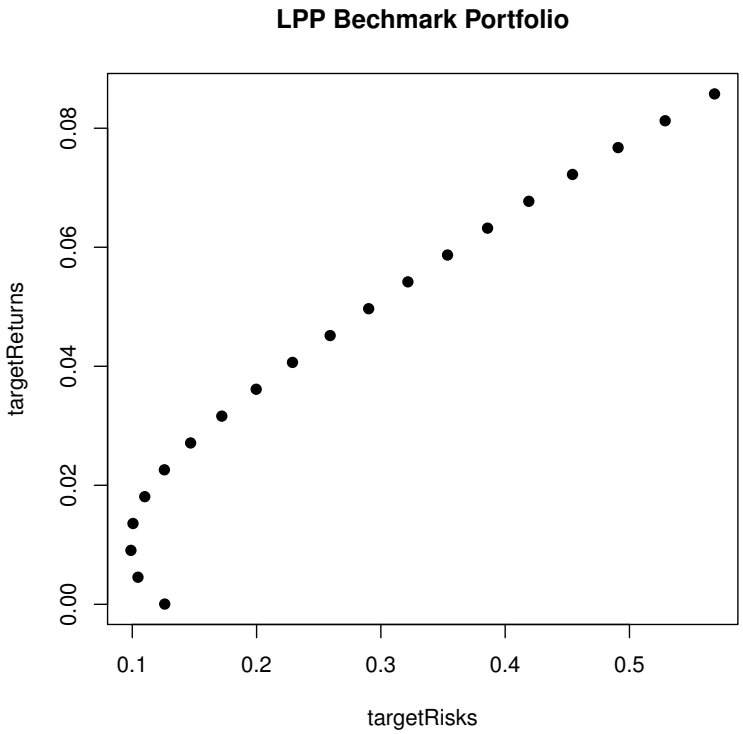

FIGURE 32.1: Efficient Frontier and Minimum Variance Locus

CHAPTER 33

MINIMUM REGRET PORTFOLIO

33.1 ASSIGNMENT

The minimum regret portfolio maximizes the minimum return for a set of return scenarios. This can be accomplished by solving the following linear program.

$$\begin{aligned} \max_{R_{min},w} \quad & R_{min} \\ s.t. \quad & \\ & w^\top \hat{\mu} = \overline{\mu} \\ & w^\top 1 = 1 \\ & w_i \geq 0 \\ & w^\top r_s - R_{min} \geq 0 \end{aligned} \tag{33.1}$$

Let us write a function which solves this optimization problem.

References

Bernd Michael Scherer and R. Douglas Martin, 2005,
Introduction to Modern Portfolio Optimization with NuOPT and S-PLUS,
Springer Publishing, New York

GNU Linear Programming Kit
http://www.gnu.org/software/glpk/glpk.html

33.2 R IMPLEMENTATION

To solve a linear optimization program with linear constraints we use R's contributed `Rglpk`, which has implemented GNU's linear programming solver tool kit. Load the library and the arguments of the solver.

```
> library(Rglpk)
Using the GLPK callable library version 4.42
> args(Rglpk_solve_LP)
function (obj, mat, dir, rhs, types = NULL, max = FALSE, bounds = NULL,
    verbose = FALSE)
NULL
```

The arguments have the following meaning

```
obj         a vector with the objective coefficients
mat         a vector or a matrix of the constraint coefficients
dir         a character vector with the directions of the constraints.
            Each element must be one of "<", "<=", ">", ">=", or "==".
rhs         the right hand side of the constraints
types       a vector indicating the types of the objective variables.
            types can be either "B" for binary, "C" for continuous or "I"
            for integer. By default all variables are of type "C".
max         a logical giving the direction of the optimization.
            TRUE means that the objective is to maximize the objective
            function, FALSE (default) means to minimize it.
bounds      NULL (default) or a list with elements upper and lower
            containing the indices and corresponding bounds of the objective
            variables. The default for each variable is a bound between
            0 and Inf.
verbose     a logical for turning on/off additional solver output,
            Default: FALSE.
```

The function returns a list with the following components:

```
solution    the vector of optimal coefficients
objval      the value of the objective function at the optimum
status      an integer with status information about the solution returned:
            0 if the optimal solution was found, a non-zero value otherwise.
```

Alternatively we can use the solver function `Rsymphony_solve_LP()` from the contributed package `Rsymphony`.
Now we are ready to write a function to optimize the minimum regret portfolio for a set of asset returns and a given target return. The function body consists of several parts: 1 defining the vector for the objective function, 2 setting up the matrix of linear constraints excluding the simple bounds, 3 creating the vectors of directions and the valus of the right hand side, and 4 setting the values for the lower and upper bounds. And the final step is the optimization itself.

```
> portfolioWeights <- function(assetReturns, targetReturn) {
     assetNames = colnames(assetReturns)
     assetReturns = as.matrix(assetReturns)
     nAssets = ncol(assetReturns)
     nScenarios = nrow(assetReturns)
     mu = colMeans(assetReturns)
     obj <- c(R_min = 1, Weights = rep(0, nAssets))
```

```
    mat = rbind(cbind(matrix(0, ncol = 1), t(mu)), cbind(matrix(0,
        ncol = 1), t(rep(1, nAssets))), cbind(matrix(rep(-1,
        nScenarios), nScenarios), -assetReturns))
    dir = c(Return = "==", Budget = "==", Scenarios = rep(">=",
        nScenarios))
    rhs = c(Returns = targetReturn, Budget = 1, Scenarios = rep(0,
        nScenarios))
    bounds = list()
    bounds$lower$ind = 1:length(obj)
    bounds$upper$ind = 1:length(obj)
    bounds$lower$val = c(Rmin = -Inf, Weights = rep(0, nAssets))
    bounds$upper$val = c(Rmin = Inf, Weights = rep(1, nAssets))
    ans = Rglpk_solve_LP(obj = obj, mat = mat, dir = dir, rhs = rhs,
        bounds = bounds, max = TRUE)
    weights = ans$solution[-1]
    names(weights) = assetNames
    weights
}
```

33.3 EXAMPLES

As an example we consider again as in the previous case study the Swiss pension fund benchmark. The data can be loaded from the Rmetrics package `fBasics`. We use daily percentage returns

```
> library(fBasics)
> assetReturns <- 100 * LPP2005REC[, 1:6]
> head(assetReturns)
GMT
                SBI       SPI      SII       LMI      MPI       ALT
2005-11-01 -0.061275  0.841460 -0.31909 -0.110888 0.154806 -0.257297
2005-11-02 -0.276201  0.251934 -0.41176 -0.117594 0.034288 -0.114160
2005-11-03 -0.115309  1.270729 -0.52094 -0.099246 1.050296  0.500744
2005-11-04 -0.323575 -0.070276 -0.11272 -0.119853 1.167956  0.948268
2005-11-07  0.131097  0.620523 -0.17958  0.036037 0.270962  0.472395
2005-11-08  0.053931  0.032926  0.21034  0.232704 0.034684  0.085362

> end(assetReturns)

GMT
[1] [2007-04-11]
```

In this example we choose the grand mean of all assets as the values for the target return.

```
> targetReturn = mean(assetReturns)
> targetReturn
[1] 0.043077
```

The next step will be the optimization of the portfolio

```
> weights = portfolioWeights(assetReturns, targetReturn)
> weights
```

```
       SBI      SPI      SII      LMI      MPI      ALT
0.000000 0.033482 0.118310 0.440168 0.000000 0.408041
```

Now compare the weights with those from the mean-variance Markowitz portfolio.

PART IX

APPENDIX

APPENDIX A

RMETRICS TERMS OF LEGAL USE

Grant of license

Rmetrics Association and *Finance Online* Zurich have authorized you to download one copy of this electronic book (ebook). The service includes free updates for the period of one year. *Rmetrics Association* and *Finance Online* Zurich grant you a nonexclusive, nontransferable license to use this eBook according to the terms and conditions specified in the following. This license agreement permits you to install and use the eBook for your personal use only.

Restrictions

You shall not resell, rent, assign, timeshare, distribute, or transfer all or any part of this eBook (including code snippets and functions) or any rights granted hereunder to any other person.
You shall not duplicate this eBook, except for backup or archival purposes. You shall not remove any proprietary notices, labels, or marks from this eBook and transfer to any other party.
The code snippets and functions provided in this book are for teaching and educational research, i.e. for non commercial use. It is not allowed to use the provided code snippets and functions for any commercial use. This includes workshops, seminars, courses, lectures, or any other events.
The unauthorized use or distribution of copyrighted or other proprietary content from this eBook is illegal.

intellectual property protection

This eBook is owned by the *Rmetrics Association* and *Finance Online* and is protected by international copyright and other intellectual property laws.

Rmetrics Association and *Finance Online* Zurich reserve all rights in this eBook not expressly granted herein. This license and your right to use this eBook terminates automatically if you violate any part of this agreement. In the event of termination, you must destroy the original and all copies of this eBook.

General

This agreement constitutes the entire agreement between you and *Rmetrics Association* and *Finance Online* Zurich and supersedes any prior agreement concerning this eBook. This agreement is governed by the laws of Switzerland.

APPENDIX B

R MANUALS ON CRAN

The R core team creates several manuals for working with R[1]
The platform dependent versions of these manuals are part of the respective R installations. They can be downloaded as PDF files from the URL given above or can directly browsed as HTML.

```
http://cran.r-project.org/manuals.html
```

The following manuals are available:

- An Introduction to R is based on the former "Notes on R", gives an introduction to the language and how to use R for doing statistical analysis and graphics.
- A draft of The R language definition documents the language per se. That is, the objects that it works on, and the details of the expression evaluation process, which are useful to know when programming R functions.
- Writing R Extensions covers how to create your own packages, write R help files, and the foreign language (C, C++, Fortran, ...) interfaces.
- R Data Import/Export describes the import and export facilities available either in R itself or via packages which are available from CRAN.
- R Installation and Administration.
- R Internals: a guide to the internal structures of R and coding standards for the core team working on R itself.
- The R Reference Index: contains all help files of the R standard and recommended packages in printable form, (approx. 3000 pages).

[1] The manuals are created on Debian Linux and may differ from the manuals for Mac or Windows on platform-specific pages, but most parts will be identical for all platforms.

The latex or texinfo sources of the latest version of these documents are contained in every R source distribution. Have a look in the subdirectory doc/manual of the extracted archive.
The HTML versions of the manuals are also part of most R installations. They are accessible using function `help.start()`.

INDEX

About the Authors

Diethelm Würtz is private lecturer at the Institute for Theoretical Physics, ITP, and for the Curriculum Computational Science and Engineering, CSE, at the Swiss Federal Institute of Technology in Zurich. He teaches Econophysics at ITP and supervises seminars in Financial Engineering at CSE. Diethelm is senior partner of Finance Online, an ETH spin-off company in Zurich, and co-founder of the Rmetrics Association.

Yohan Chalabi has a master in Physics from the Swiss Federal Institute of Technology in Lausanne. He is now a PhD student in the Econophysics group at ETH Zurich at the Institute for Theoretical Physics. Yohan is a co-maintainer of the Rmetrics packages.

Longhow Lam Longhow currently works for ABNAMRO where he builds credit risk models. He has a lot of practical experience in using R / S-PLUS and SAS in predictive model building for different companies. He has given many introductory courses on the S language.

Andrew Ellis read neuroscience and mathematics at the University in Zurich. He works for Finance Online and is currently doing an internship in the Econophysics group at ETH Zurich. Andrew is working on the Rmetrics documentation project.