Statistical Analysis with R

- a quick start -

OLEG NENADIĆ, WALTER ZUCCHINI

September 2004

Contents

Preface

This introduction to the freely available statistical software package **R** is primarily intended for people already familiar with common statistical concepts. Thus the statistical methods used to illustrate the package are not explained in detail. These notes are not meant to be a reference manual, but rather a hands-on introduction for statisticians who are unfamiliar with **R**. The intention is to offer just enough material to get started, to motivate beginners by illustrating the power and flexibility of **R**, and to show how simply it enables the user to carry out sophisticated statistical computations and to produce high-quality graphical displays.

The notes comprise four sections, which build on each other and should therefore be read sequentially. The first section *(An Introduction to* **R***)* introduces the most basic concepts. Occasionally things are simplified and restricted to the minimum background in order to avoid obscuring the main ideas by offering too much detail. The second and the third section *(Linear Models* and *Time Series Analysis)* illustrate some standard **R** commands pertaining to these two common statistical topics. The fourth section *(Advanced Graphics)* covers some of the excellent graphical capabilities of the package.

Throughout the text `typewriter` font is used for annotating **R** functions and options. **R** functions are given with brackets, e.g. `plot()` while options are typed in italic typewriter font, e.g. *`xlab="x label"`*. **R** commands which are entered by the user are printed in red and the output from **R** is printed in blue. The datasets used are available from the URI *http://134.76.173.220/R_workshop*.

An efficient (and enjoyable) way of beginning to master **R** is to actively use it, to experiment with it's functions and options and to write own functions. It is not necessary to study lengthy manuals in order to get started; one can get useful work done almost immediately. Thus, the main goal of this introduction is to motivate the reader to actively explore **R**. Good luck!

Chapter 1

An Introduction to R

1.1 Downloading and Installing R

R is a widely used environment for statistical analysis. The striking difference between **R** and most other statistical packages is that it is free software and that it is maintained by scientists for scientists. Since its introduction in 1996 by R. Ihaka and R. Gentleman, the **R** project has gained many users and contributors who continuously extend the capabilities of **R** by releasing add–ons (packages) that offer new functions and methods, or improve the existing ones.
One disadvantage or advantage, depending on the point of view, is that **R** is used within a command–line interface, which imposes a slightly steeper learning curve than other software. But, once this hurdle has been overcome, **R** offers almost unlimited possibilities for statistical data analysis.

R is distributed by the "Comprehensive R Archive Network" (CRAN) – it is available from the URI: *http://cran.r-project.org*. The current version of **R** (1.9.1 as of September 2004, approx. 20 MB) for Windows can be downloaded by selecting "*R binaries*" → "*windows*" → "*base*" and downloading the file "*rw1091.exe*" from the CRAN–website. **R** can then be installed by executing the downloaded file. The installation procedure is straightforward; one usually only has to specify the target directory in which to install **R**. After the installation, **R** can be started like any other application for Windows, that is by double–clicking on the corresponding icon.

1.2 Getting Started

Since **R** is a command line based language, all commands are entered directly into the console. A starting point is to use **R** as a substitute for a pocket calculator. By typing

```
2+3
```

into the console, **R** adds 3 to 2 and displays the result. Other simple operators include

```
2-3         # Subtraction
2*3         # Multiplication
2/3         # Division
2^3         # 2^3
sqrt(3)     # Square roots
log(3)      # Logarithms (to the base e)
```

Operators can also be nested, e.g.

```
(2 - 3) * 3
```

first subtracts 3 from 2 and then multiplies the result with 3.
Often it can be useful to store results from operations for later use. This can be done using the "assignment operator" `<-` , e.g.

`<-`

```
test <- 2 * 3
```

performs the operation on the right hand side (`2*3`) and then stores the result as an object named `test`. (One can also use `=` or even `->` for assignments.) Further operations can be carried out on objects, e.g.

```
2 * test
```

multiplies the value stored in `test` with 2. Note that objects are overwritten without notice. The command `ls()` outputs the list of currently defined objects.

`ls()`

Data types

As in other programming languages, there are different data types available in **R**, namely *"numeric"*, *"character"* and *"logical"*. As the name indicates, "numeric" is used for numerical values (double precision). The type "character" is used for characters and is generally entered using quotation marks:

```
myname <- "what"
myname
```

However, it is not possible (nor meaningful) to apply arithmetic operators on character data types. The data type "logical" is used for boolean variables: (`TRUE` or `T`, and `FALSE` or `F`).

Object types

Depending on the structure of the data, **R** recognises 4 standard object types: *"vectors"*, *"matrices"*, *"data frames"* and *"lists"*. Vectors are one–dimensional arrays of data; matrices are two–dimensional data arrays. Data frames and lists are further generalizations and will be covered in a later section.

Creating vectors in **R**
There are various means of creating vectors in **R**. E.g. in case one wants to save the numbers $3, 5, 6, 7, 1$ as `mynumbers`, one can use the `c()` command:

`c()`

```
mynumbers <- c(3, 5, 6, 7, 1)
```

Further operations can then be carried out on the **R** object `mynumbers`. Note that arithmetic operations on vectors (and matrices) are carried out component–wise, e.g. `mynumbers*mynumbers` returns the squared value of each component of `mynumbers`.
Sequences can be created using either `":"` or `seq()`:

`:`

```
1:10
```

creates a vector containing the numbers $1, 2, 3, \ldots, 10$. The `seq()` command allows the increments of the sequence to be specified:

`seq()`

```
seq(0.5, 2.5, 0.5)
```

creates a vector containing the numbers $0.5, 1, 1.5, 2, 2.5$. Alternatively one can specify the length of the sequence:

```
seq(0.5, 2.5, length = 100)
```

creates a sequence from 0.5 to 2.5 with the increments chosen such that the resulting sequence contains 100 equally spaced values.

Creating matrices in **R**
One way of creating a matrix in **R** is to convert a vector of length $n \cdot m$ into a $n \times m$ matrix:

```
mynumbers <- 1:12
matrix(mynumbers, nrow = 4)
```

`matrix()`

Note that the matrix is created columnwise – for rowwise construction one has to use the option `byrow=TRUE`:

```
matrix(mynumbers, nrow = 4, byrow = TRUE)
```

An alternative way for constructing matrices is to use the functions `cbind()` and `rbind()`, which combine vectors (row- or columnwise) to a matrix:

```
mynumbers1 <- 1:4
mynumbers2 <- 11:14
cbind(mynumbers1, mynumbers2)
rbind(mynumbers1, mynumbers2)
```

cbind()

rbind()

Accessing elements of vectors and matrices
Particular elements of **R** vectors and matrices can be accessed using square brackets. Assume that we have created the following **R** objects `vector1` and `matrix1`:

```
vector1 <- seq(-3, 3, 0.5)
matrix1 <- matrix(1:20, nrow = 5)
```

Some examples of how to access particular elements are given below:

```
vector1[5]          # returns the 5th element of vector1
vector1[1:3]        # returns the first three elements of vector1
vector1[c(2, 4, 5)] # returns the 2nd, 4th and 5th element of
                      vector1
vector1[-5]         # returns all elements except for the 5th one
```

Elements of matrices are accessed in a similar way. `matrix1[a,b]` returns the value from the a–th row and the b–th column of `matrix1`:

```
matrix1[2,]         # returns the 2nd row of matrix1
matrix1[,3]         # returns the 3rd column of matrix1
matrix1[2, 3]       # returns the value from matrix1 in the
                      2nd row and 3rd column
matrix1[1:2, 3]     # returns the value from matrix1 in the first
                      two rows and the 3rd column
```

Example: Plotting functions
Assume that you were to plot a function by hand. One possibility of doing it is to

1. Select some x–values from the range to be plotted
2. Compute the corresponding $y = f(x)$ values
3. Plot x against y
4. Add a (more or less) smooth line connecting the (x, y)–points

Graphs of functions are created in essentially the same way in **R**, e.g. plotting the function $f(x) = \sin(x)$ in the range of $-\pi$ to π can be done as follows:

```
x <- seq(-pi, pi, length = 10)  # defines 10 values from −π to π
y <- sin(x)                     # computes the corresponding
                                  y–values
plot(x, y)                      # plots x against y
lines(x, y)                     # adds a line connecting the
                                  (x, y)–points
```

`plot()`

`lines()`

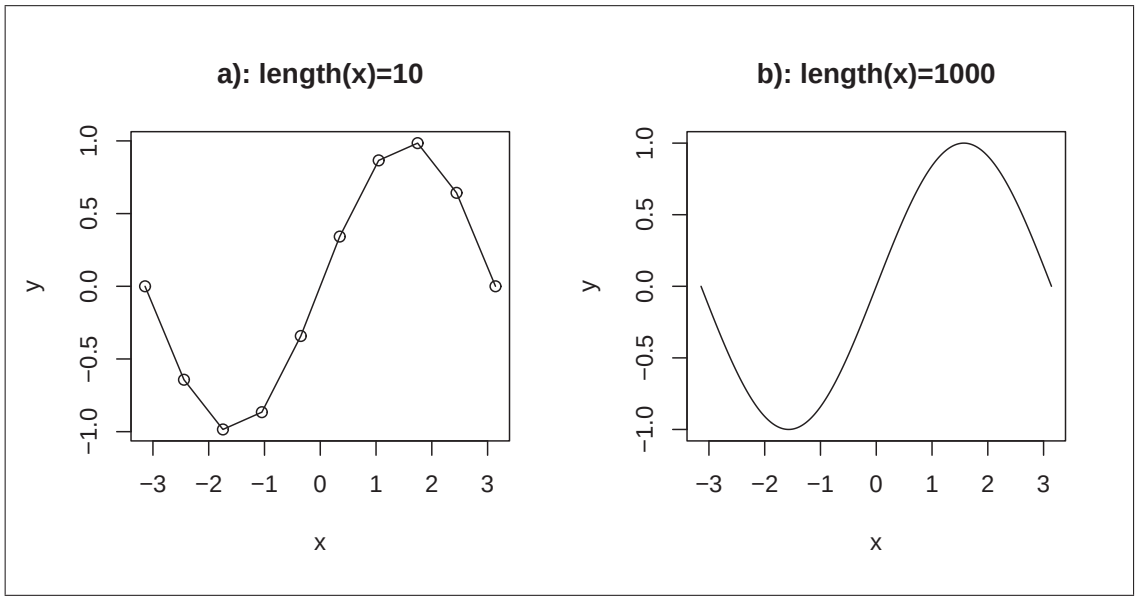

Figure 1.1: *Plotting $sin(x)$ in* **R**.

The output is shown in the left part of figure 1.1. However, the graph does not look very appealing since it lacks smoothness. A simple "trick" for improving the graph is to simply increase the number of x–values at which $f(x)$ is evaluated, e.g. to 1000:

```
x <- seq(-pi, pi, length = 1000)
y <- sin(x)
plot(x, y, type = "l")
```

The result is shown in the right part of figure 1.1. Note the use of the option `type="l"`, which causes the graph to be drawn with connecting lines rather than points.

1.3 Statistical Distributions

The names of the **R** functions for distributions comprise two parts. The first part (the first letter) indicates the "function group", and the second part (the remainder of the function name) indicates the distribution. The following "function groups" are available:

- probability density function (`d`)
- cumulative distribution function (`p`)
- quantile function (`q`)
- random number generation (`r`)

Common distributions have their corresponding **R** "names":

distribution	**R** name	*distribution*	**R** name	*distribution*	**R** name
normal	`norm`	t	`t`	χ^2	`chisq`
exponential	`exp`	f	`f`	uniform	`unif`
log-normal	`lnorm`	beta	`beta`	gamma	`gamma`
logistic	`logis`	weibull	`weibull`	cauchy	`cauchy`
geometric	`geom`	binomial	`binom`	hypergeometric	`hyper`
poisson	`pois`	negative binomial	`nbinom`		

E.g., random numbers (`r`) from the normal distribution (`norm`) can be drawn using the `rnorm()` function; quantiles (`q`) of the χ^2–distribution (`chisq`) are obtained with `qchisq()`.
The following examples illustrate the use of the **R** functions for computations involving statistical distributions:

```
rnorm(10)          # draws 10 random numbers from a standard
                     normal distribution
rnorm(10, 5, 2)    # draws 10 random numbers from a N(μ = 5,σ = 2)
                     distribution
pnorm(0)           # returns the value of a standard normal cdf at t = 0
qnorm(0.5)         # returns the 50% quantile of the standard normal
                     distribution
```

Examples for handling distributions
Assume that we want to generate 50 (standard) normally distributed random numbers and to display them as a histogram. Additionally, we want to add the pdf of the ("fitted") normal distribution to the plot as shown in figure 1.2:

```
mysample <- rnorm(50)            # generates random numbers
hist(mysample, prob = TRUE)      # draws the histogram
mu <- mean(mysample)             # computes the sample mean
sigma <- sd(mysample)            # computes the sample standard
                                   deviation
x <- seq(-4, 4, length = 500)    # defines x–values for the pdf
y <- dnorm(x, mu, sigma)         # computes the normal pdf
lines(x, y)                      # adds the pdf as "lines" to the plot
```

hist()

mean()

sd()

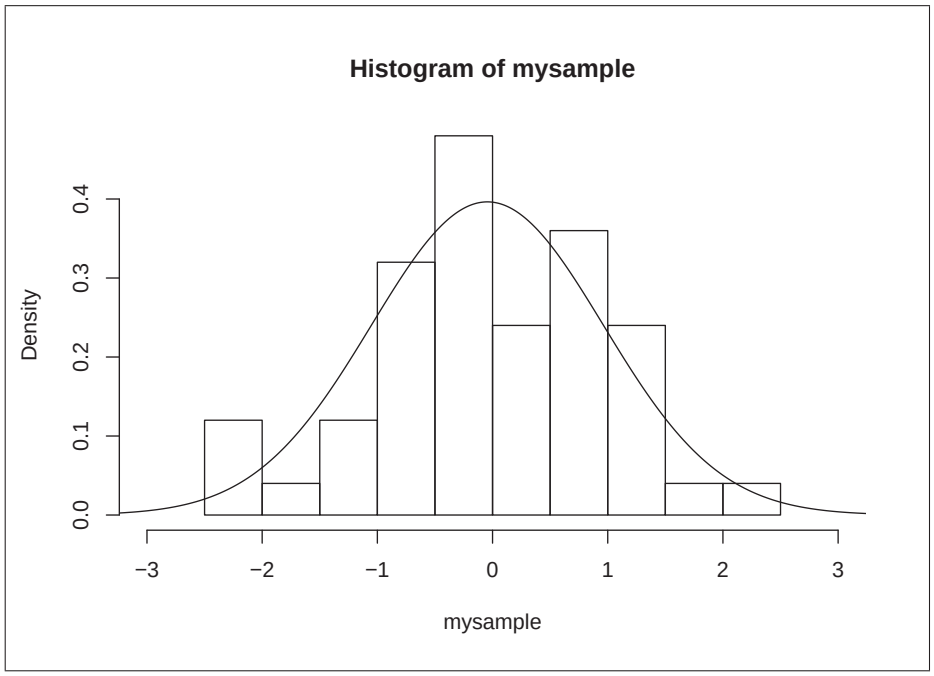

Figure 1.2: *Histogram of normally distributed random numbers and "fitted" density.*

Another example (figure 1.3) is the visualization of the approximation of the binomial distribution with the normal distribution for e.g. $n = 50$ and $\pi = 0.25$:

```
x <- 0:50                        # defines the x–values
y <- dbinom(x, 50, 0.25)         # computes the binomial
                                   probabilities
plot(x, y, type="h")             # plots binomial probabilities
x2 <- seq(0, 50, length = 500)   # defines x–values (for the
                                   normal pdf)
y2 <- dnorm(x2, 50*0.25,
      sqrt(50*0.25*(1-0.25)))    # computes the normal pdf
lines(x2, y2, col = "red")       # draws the normal pdf
```

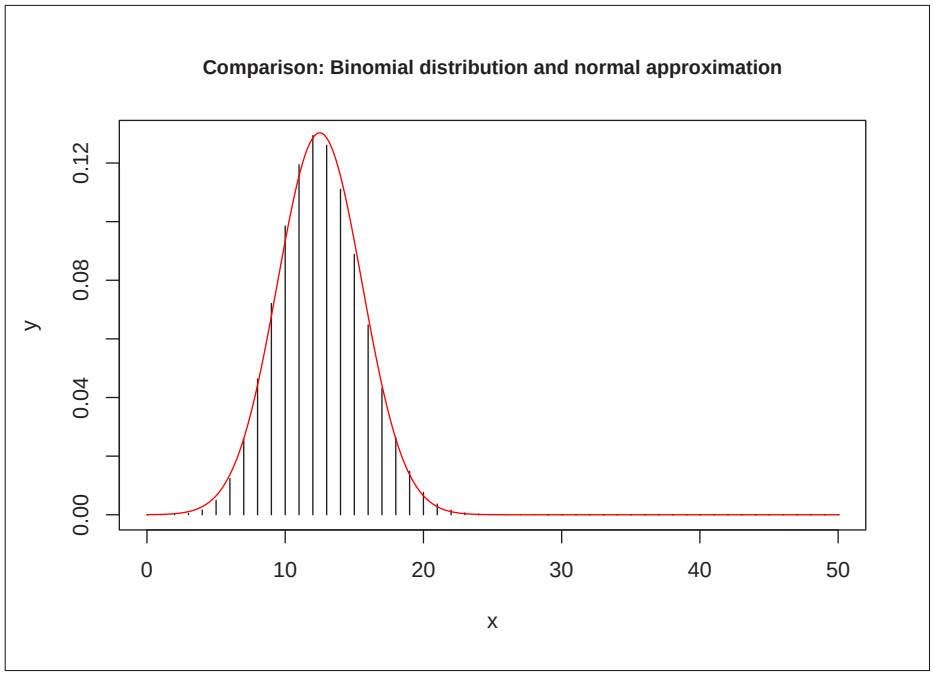

Figure 1.3: *Comparing the binomial distribution with* $n = 50$ *and* $\pi = 0.25$ *with an approximation by the normal distribution* ($\mu = n \cdot \pi$, $\sigma = \sqrt{n \cdot \pi \cdot (1 - \pi)}$).

1.4 Writing Custom R Functions

In case **R** does not offer a required function, it is possible to write a custom one. Assume that we want to compute the geometric mean of a sample:

$$\mu_G = \prod_{i=1}^{n} x_i \, ^{\frac{1}{n}} = e^{\frac{1}{n} \sum_i \log(x_i)}$$

Since **R** doesn't have a function for computing the geometric mean, we have to write our own function `geo.mean()`:

```
fix(geo.mean)
```

`fix()`

opens an editor window where we can enter our function:

```
function(x)
{
  n <- length(x)
  gm <- exp(mean(log(x)))
  return(gm)
}
```

function()

Note that **R** checks the function after closing and saving the editor–window. In case of "structural" errors (the most common case for that are missing brackets), **R** reports these to the user. In order to fix the error(s), one has to enter

```
geo.mean <- edit()
```

edit()

since the (erroneous) results are not saved. (Using `fix(geo.mean)` results in loosing the last changes.)

Chapter 2

Linear Models

2.1 Fitting Linear Models in R

This section focuses on the three "main types" of linear models: Regression, Analysis of Variance and Analysis of Covariance.

Simple regression analysis

The dataset *"strength"*, which is stored as *'strength.dat'*, contains measurements on the ability of workers to perform physically demanding tasks. It contains the measured variables *"grip"*, *"arm"*, *"rating"* and *"sims"* collected from 147 persons. The dataset can be imported into **R** with

```
strength <- read.table("C:/R_workshop/strength.dat",
                       header = TRUE)
```

`read.table()`

The command `read.table()` reads a file into **R** assuming that the data is structured as a matrix (table). It assumes that the entries of a row are separated by blank spaces (or any other suitable separator) and the rows are separated by line feeds. The option `header=TRUE` tells **R** that the first row is used for labelling the columns.
In order to get an overview over the relation between the 4 (quantitative) variables, one can use

```
pairs(strength)
```

`pairs()`

which creates a matrix of scatterplots for the variables.
Let's focus on the relation between *"grip"* (1st column) and *"arm"* (2nd column). The general function for linear models is `lm()`. Fitting the model

$$\text{grip}_i = \beta_0 + \beta_1 \cdot \text{arm}_i + e_i$$

can be done using

```
fit <- lm(strength[,1]~strength[,2])
fit
```

lm()

The function `lm()` returns a list object which we have saved under some name, e.g. as `fit`. As previously mentioned, lists are a generalized object–type; a list can contain several objects of different types and modes arranged into a single object. The names of the entries stored in a list can be viewed using

```
names(fit)
```

names()

One entry in this list is `coefficients` which contains the coefficients of the fitted model. The coefficients can be accessed using the "$"–sign:

```
fit$coefficients
```

returns a vector (in this case of length 2) containing the estimated parameters ($\hat{\beta}_0$ and $\hat{\beta}_1$). Another entry is `residuals`, which contains the residuals of the fitted model:

```
res <- fit$residuals
```

Before looking further at our fitted model, let us briefly examine the residuals. A first insight is given by displaying the residuals as a histogram:

```
hist(res, prob = TRUE, col = "red")
```

hist()

An alternative is to use a kernel density estimate and to display it along with the histogram:

```
lines(density(res), col = "blue")
```

The function `density()` computes the kernel density estimate (other methods for kernel density estimation will be discussed in a later section).

density()

Here one might also wish to add the pdf of the normal distribution to the graph:

```
mu <- mean(res)
sigma <- sd(res)
x <- seq(-60, 60, length = 500)
y <- dnorm(x, mu, sigma)
lines(x, y, col = 6)
```

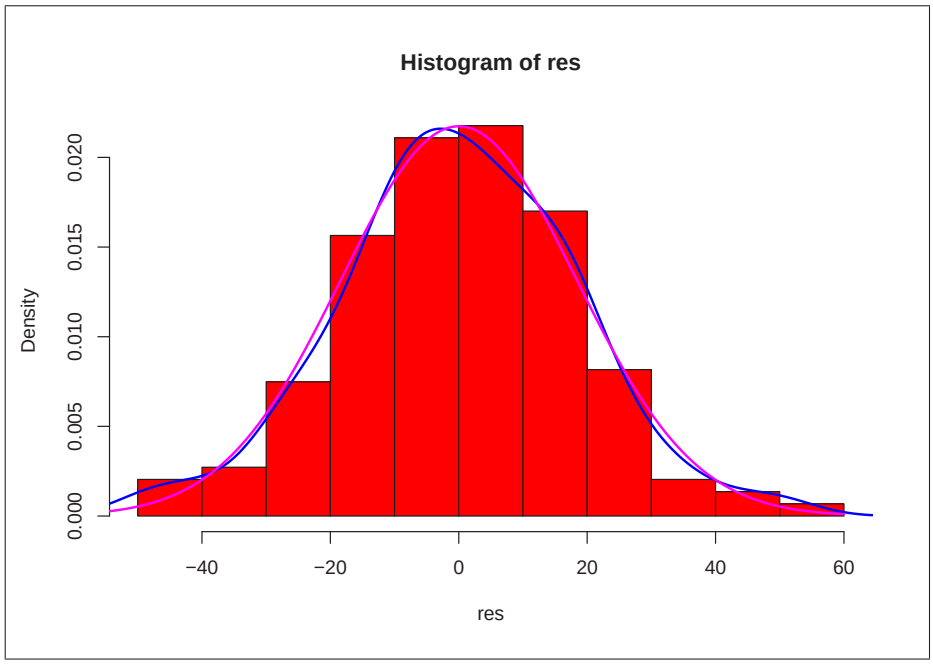

Figure 2.1: *Histogram of the model residuals with kernel density estimate and fitted normal distribution.*

There also exist alternative ways for (graphically) investigating the normality of a sample, for example QQ-plots:

```
qqnorm(res)
```

qqnorm()

draws the sample quantiles against the quantiles of a normal distribution as shown in figure 2.2. Without going into detail, the ideal case is given when the points lie on a straight line.

Another option is to specifically test for normality, e.g. using the Kolmogorov-Smirnov test or the Shapiro-Wilks test:

```
ks.test(res, "pnorm", mu, sigma)
```

ks.test()

performs the Kolmogorov-Smirnov test on `res`. Since this test can be used for any distribution, one has to specify the distribution (`pnorm`) and its parameters (`mu` and `sigma`). The Shapiro–Wilks test specifically tests for normality, so one only has to specify the data:

```
shapiro.test(res)
```

shapiro.test()

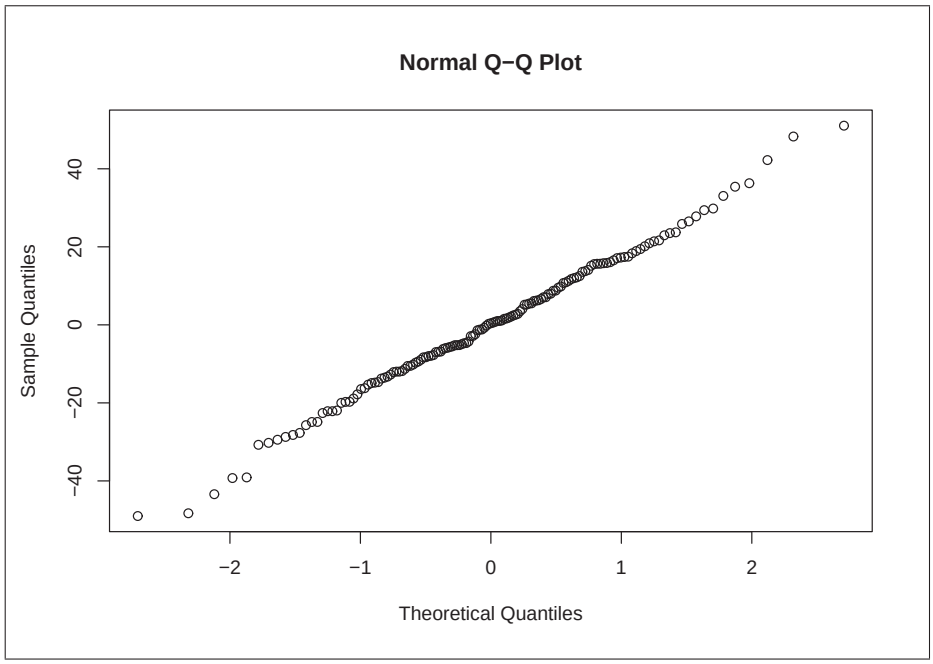

Figure 2.2: *QQ-plot of residuals.*

Now back to our fitted model. In order to display the observations together with the fitted model, one can use the following code which creates the graph shown in figure 2.3:

```
plot(strength[,2], strength[,1])
betahat <- fit$coefficients
x <- seq(0, 200, length = 500)
y <- betahat[1] + betahat[2]*x
lines(x, y, col = "red")
```

Another useful function in this context is `summary()`: `summary()`

```
summary(fit)
```

returns an output containing the values of the coefficients and other information:

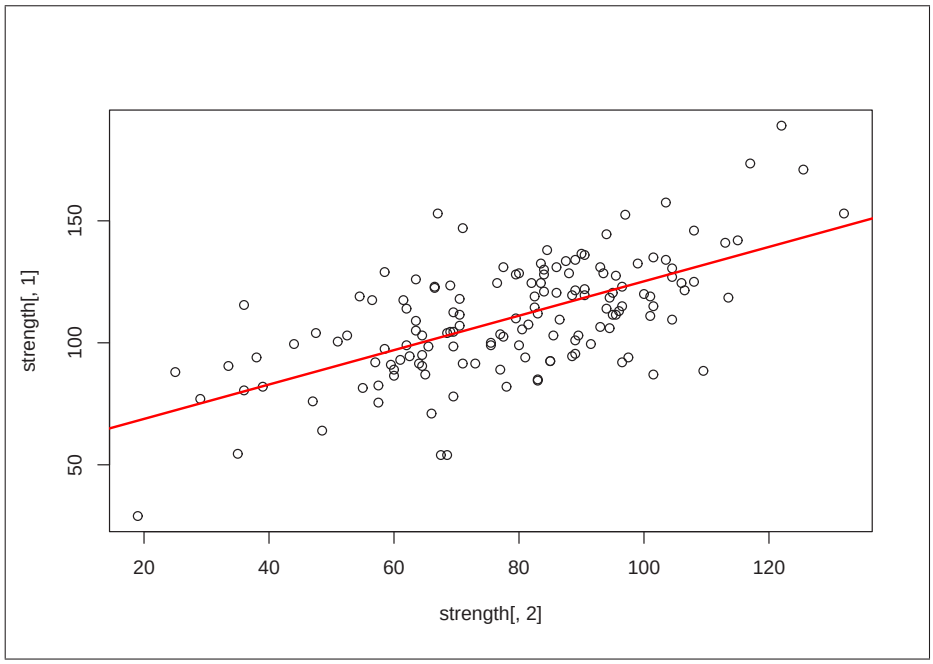

Figure 2.3: *Observations (*`strength[,2]` *vs.* `strength[,1]`*) and fitted line.*

```
Call:
lm(formula = strength[, 1] ~ strength[, 2])

Residuals:
     Min       1Q   Median       3Q      Max
-49.0034 -11.5574   0.4104  12.3367  51.0541

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   54.70811    5.88572   9.295   <2e-16 ***
strength[, 2]  0.70504    0.07221   9.764   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 18.42 on 145 degrees of freedom
Multiple R-Squared: 0.3967,     Adjusted R-squared: 0.3925
F-statistic: 95.34 on 1 and 145 DF,  p-value: < 2.2e-16
```

Analysis of Variance

Consider the dataset *'miete.dat'*, which contains rent prices for apartments in a German city. Two factors were also recorded: year of construction, and whether the apartment was on the ground floor, first floor, ..., fourth floor. Again, the data can be imported into **R** using the `read.table()` command:

```
rent <- read.table("C:/R_workshop/miete.dat", header = TRUE)
```

In this case, `rent` is a matrix comprising three columns: The first one *("Baujahr")* indicates the year of construction, the second one *("Lage")* indicates the floor and the third column contains the rent prices *("Miete")* per a square meter. We can start by translating the German labels into English:

```
names(rent)
```

`names()`

returns the names of the `rent` objects. The names can be changed by typing

```
names(rent) <- c("year", "floor", "price")
```

into the console.
In order to examine the relationship between price and floor, one can use boxplots for visualization. In order to do so, one needs to "extract" the rent prices for each floor-group:

```
price <- rent[,3]
fl <- rent[,2]
levels(fl)
```

`levels()`

Here we have saved the third column of rent as `price` and the second one as `fl`. The command `levels(fl)` shows us the levels of `fl` ("a" to "e"). It is possible to perform queries using square brackets, e.g.

```
price[fl=="a"]
```

returns the prices for the apartments on floor "a" (ground floor in this case). Accordingly,

```
fl[price<7]
```

returns the floor levels whose corresponding rent prices (per m^2) are less than 7 (Euro). These queries can be further expanded using logical AND (&) or OR (|) operators:

```
fl[price<7 & price>5]
```

returns all floor levels whose corresponding rent prices are between 5 and 7 (Euro).

A convenient function is `split(a,b)`, which splits the data `a` by the levels given in `b`. This can be used together with the function `boxplot()`:

```
boxplot(split(price, fl))
```

boxplot()

split()

Accordingly, the relation between the year of construction and the price can be visualized with

```
year <- rent[,1]
boxplot(split(price, year))
```

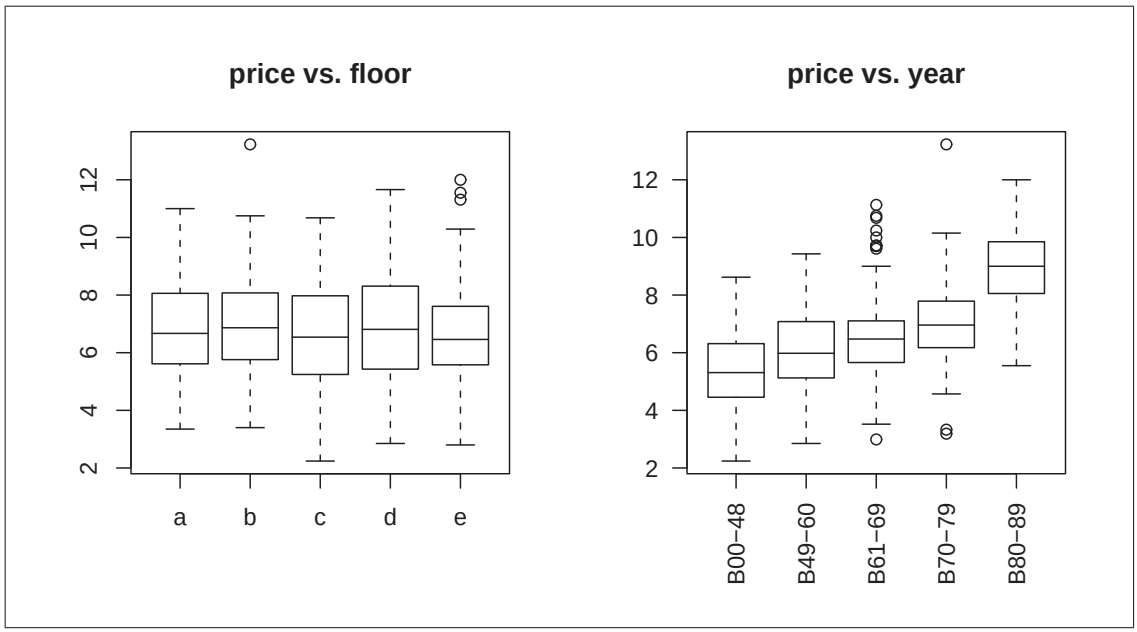

Figure 2.4: *Boxplots of the rent example. The left boxplot displays the relation between price and floor; the right boxplot shows the relation between price and year.*

The analysis of variance can be carried out in two ways, either by treating it as a linear model (`lm()`) or by using the function `aov()`, which is more convenient in this case:

```
fit1a <- lm(price~fl)
summary(fit1a)
```

lm()

returns

```
Call:
lm(formula = price ~ fl)

Residuals:
    Min      1Q  Median      3Q     Max
-4.4132 -1.2834 -0.1463  1.1717  6.2987

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   6.8593     0.1858  36.925   <2e-16 ***
flb           0.0720     0.2627   0.274    0.784
flc          -0.2061     0.2627  -0.785    0.433
fld           0.0564     0.2627   0.215    0.830
fle          -0.1197     0.2627  -0.456    0.649
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 1.858 on 495 degrees of freedom
Multiple R-Squared: 0.003348,   Adjusted R-squared: -0.004706
F-statistic: 0.4157 on 4 and 495 DF,  p-value: 0.7974
```

On the other hand,

```
fit1b <- aov(price~fl)
summary(fit1b)
```

aov()

returns

```
             Df  Sum Sq Mean Sq F value Pr(>F)
fl            4    5.74    1.43  0.4157 0.7974
Residuals   495 1708.14    3.45
```

The "full" model (i.e. including year and floor as well as interactions) is analysed with

```
fit2 <- aov(price~fl+year+fl*year)
summary(fit2)
```

```
             Df Sum Sq Mean Sq F value Pr(>F)
fl            4   5.74    1.43  0.7428 0.5632
year          4 735.26  183.81 95.1808 <2e-16 ***
fl:year      16  55.56    3.47  1.7980 0.0288 *
Residuals   475 917.33    1.93
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The interpretation of the tables is left to the reader.

Analysis of covariance
The extension to the analysis of covariance is straightforward. The dataset `car` is based on data provided by the U.S. Environmental Protection Agency (82 cases). It contains the following variables:

- `BRAND` Car manufacturer
- `VOL` Cubic feet of cab space
- `HP` Engine horsepower
- `MPG` Average miles per gallon
- `SP` Top speed (mph)
- `WT` Vehicle weight (10 lb)

Again, the data is imported using

```
car <- read.table("C:/R_workshop/car.dat", header = TRUE)

attach(car)
```

`attach()`

The `attach` command makes it possible to access columns of `car` by simply entering their name. The first column can be accessed by either typing `BRAND` or `car[,1]` into the console.
The model

$$MPG_{ijk} = \mu + \alpha_i + \theta SP_j + e_{ijk} \quad ; \ \alpha_i : \text{ Effect of BRAND i}$$

can be analysed in **R** with

```
fit3 <- aov(MPG~BRAND+SP)
summary(fit3)
```

`aov()`
`summary()`

```
            Df Sum Sq Mean Sq F value    Pr(>F)
BRAND       28 6206.7   221.7  8.6259 2.056e-11 ***
SP           1  564.5   564.5 21.9667 2.038e-05 ***
Residuals   52 1336.3    25.7
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

2.2 Generalized Linear Models

Generalized linear models enable one to model response variables that follow any distribution from the exponential family. The **R** function `glm()` fits generalized linear models. The model formula is specified in the same way as in `lm()`

`glm()`

or `aov()`. The distribution of the response needs to be specified, as does the link function, which expresses the monotone function of the conditional expectation of the response variable that is to be modelled as a linear combination of the covariates.
In order to obtain help for an R-function, one can use the built–in help–system of **R**:

`?glm` or `help(glm)` `?` `help()`

Typically, the help–document contains information on the structure of the function, an explanation of the arguments, references, examples etc. In case of `glm()`, there are several examples given. The examples can be examined by copying the code and pasting it into the **R** console. For generalized linear models, the information retrieved by

```
?family
```

`family`

is also relevant since it contains further information about the specification of the error distribution and the link function.
The following example of how to use the `glm()`– function is given in the help–file:

```
## Dobson (1990) Page 93: Randomized Controlled Trial :
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
print(d.AD <- data.frame(treatment, outcome, counts))
glm.D93 <- glm(counts ~ outcome + treatment,
               family=poisson())
anova(glm.D93)
summary(glm.D93)
```

The first three lines are used to create an **R** object with the data. The fourth line (`print()`) displays the created data; the fitting is done in the fifth line with `glm()`. The last two lines (`anova()` and `summary()`) are used for displaying the results.

2.3 Extensions

An important feature of **R** is its extension system. Extensions for **R** are delivered as packages ("libraries"), which can be loaded within **R** using the `library()` command. Usually, packages contain functions, datasets, help files and other files such as dlls (Further information on creating custom packages for **R** can be found on the **R** website). `library()`
There exist a number of packages that offer extensions for linear models. The package `mgcv` contains functions for fitting generalized additive models (`gam()`); routines for nonparametric density estimation and nonparametric regression are

offered by the `sm` package. An overview over the available **R** packages is given at *http://cran.r-project.org/src/contrib/PACKAGES.html*.

For example, fitting a GAM to the `car`– dataset can be carried out as follows:

```
library(mgcv)
fit <- gam(MPG~s(SP))
summary(fit)
```

`gam()`

The first line loads the package `mgcv` which contains the function `gam()`. In the second line the variable MPG (Miles per Gallon) was modelled as a "smooth function" of SP (Speed). Note that the structure of GAM formulae is almost identical to the standard ones in **R** – the only difference is the use of `s()` for indicating smooth functions. A summary of the fitted model is again given by the `summary()`– command.
Plotting the observations and the fitted model as shown in figure 2.5 can be done in the following way:

```
plot(HP, MPG)
x <- seq(0, 350, length = 500)
y <- predict(fit, data.frame(HP = x))
lines(x, y, col = "red", lwd = 2)
```

In this case, the (generic) function `predict()` was used for "predicting" (i.e. obtaining $\hat{y}$ at the specified values of the covariate, here x).

`predict()`

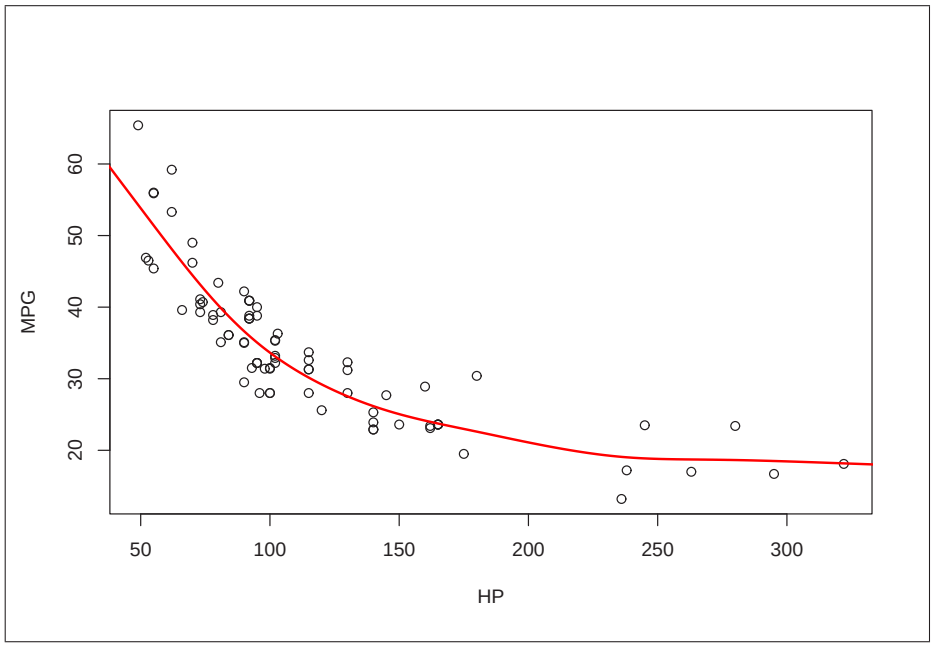

Figure 2.5: *Fitting a "simple" GAM to the* `car` *data.*

Chapter 3

Time Series Analysis

3.1 Classical Decomposition

Linear Filtering of Time Series

A key concept in traditional time series analysis is the decomposition of a given time series X_t into a trend T_t, a seasonal component S_t and the remainder or residual, e_t.
A common method for obtaining the trend is to use linear filters on given time series:

$$T_t = \sum_{i=-\infty}^{\infty} \lambda_i X_{t+i}$$

A simple class of linear filters are moving averages with equal weights:

$$T_t = \frac{1}{2a+1} \sum_{i=-a}^{a} X_{t+i}$$

In this case, the filtered value of a time series at a given period τ is represented by the average of the values $\{x_{\tau-a}, \ldots, x_\tau, \ldots, x_{\tau+a}\}$. The coefficients of the filtering are $\{\frac{1}{2a+1}, \ldots, \frac{1}{2a+1}\}$.

Consider the dataset `tui`, which contains stock data for the TUI AG from Jan., 3rd 2000 to May, 14th 2002, namely date (1st column), opening values (2nd column), highest and lowest values (3rd and 4th column), closing values (5th column) and trading volumes (6th column). The dataset has been exported from Excel © as a CSV–file (comma separated values). CSV–files can be imported into **R** with the function `read.csv()`:

`read.csv()`

```
tui <- read.csv("C:/R_workshop/tui.csv", header = TRUE,
                dec = ",", sep = ";")
```

The option `dec` specifies the decimal separator (in this case, a comma has been used as a decimal separator. This option is not needed when a dot is used as a

decimal separator.) The option `sep` specifies the separator used to separate entries of the rows.
Applying simple moving averages with $a = 2, 12,$ and 40 to the closing values of the `tui`– dataset implies using following filters:

- $a = 2 : \lambda_i = \{\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}\}$
- $a = 12 : \lambda_i = \underbrace{\{\frac{1}{25}, \ldots, \frac{1}{25}\}}_{25 \text{ times}}$
- $a = 40 : \lambda_i = \underbrace{\{\frac{1}{81}, \ldots, \frac{1}{81}\}}_{81 \text{ times}}$

The resulting filtered values are (approximately) weekly ($a = 2$), monthly ($a = 12$) and quarterly ($a = 40$) averages of returns. Filtering is carried out in **R** with the `filter()`– command.

`filter()`

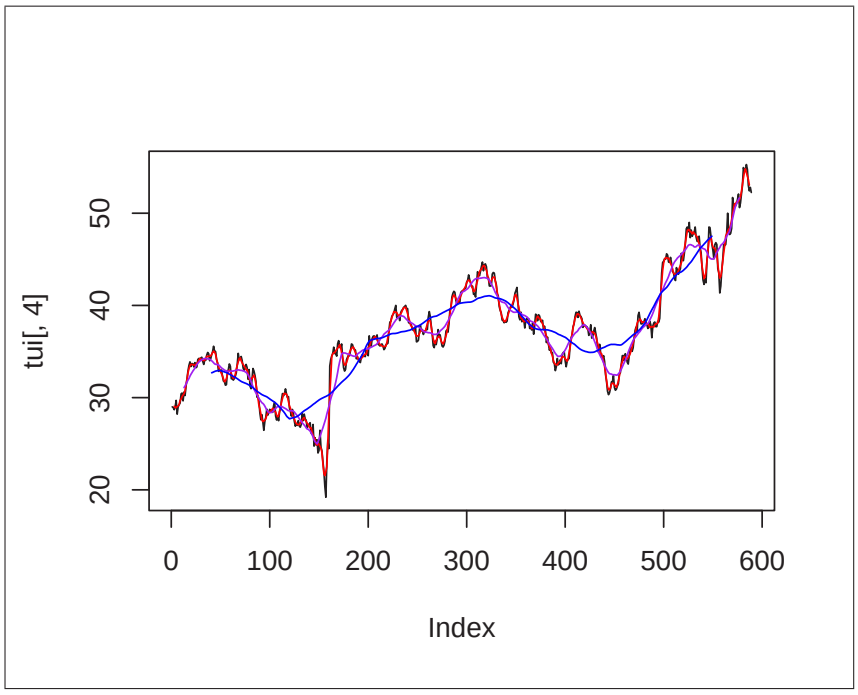

Figure 3.1: *Closing values and averages for* $a = 2, 12$ *and* 40.

The following code was used to create figure 3.1 which plots the closing values of the TUI shares and the averages, displayed in different colours.

```
plot(tui[,5], type = "l")
  tui.1 <- filter(tui[,5], filter = rep(1/5, 5))
```

```
  tui.2 <- filter(tui[,5], filter = rep(1/25, 25))
  tui.3 <- filter(tui[,5], filter = rep(1/81, 81))
lines(tui.1, col = "red")
lines(tui.2, col = "purple")
lines(tui.3, col = "blue")
```

Decomposition of Time Series

Another possibility for evaluating the trend of a time series is to use a nonparametric regression technique (which is also a special type of linear filter). The function `stl()` performs a seasonal decomposition of a given time series X_t by determining the trend T_t using "loess" regression and then computing the seasonal component S_t (and the residuals e_t) from the differences $X_t - T_t$.

stl()

Performing the seasonal decomposition for the time series `beer` (monthly beer production in Australia from Jan. 1956 to Aug. 1995) is done using the following commands:

```
beer <- read.csv("C:/R_workshop/beer.csv", header = TRUE,
                 dec = ",", sep = ";")
beer <- ts(beer[,1], start = 1956, freq = 12)
plot(stl(log(beer), s.window = "periodic"))
```

The data is read from `C:/R_workshop/beer.csv` and then transformed with `ts()` into a `ts`– object. This "transformation" is required for most of the time series functions, since a time series contains more information than the values itself, namely information about dates and frequencies at which the time series has been recorded.

ts()

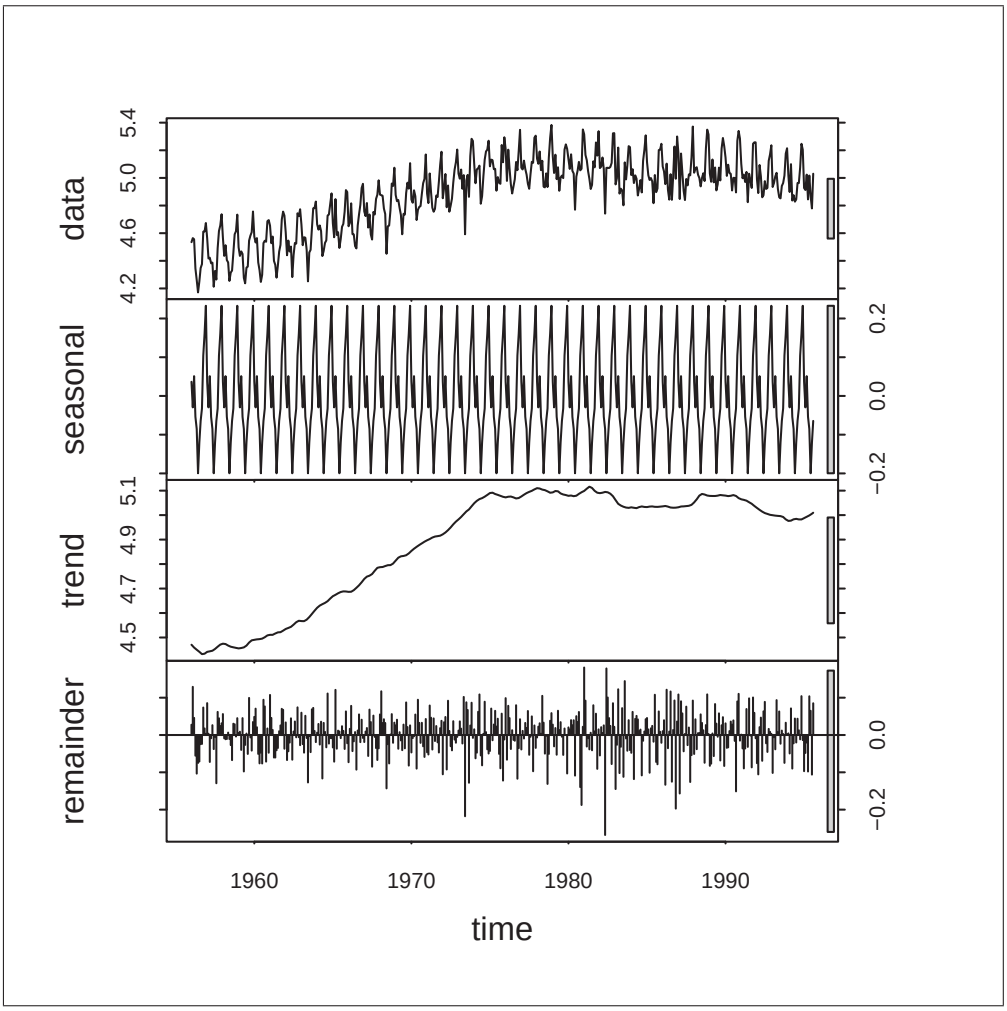

Figure 3.2: *Seasonal decomposition using* `stl()`.

Regression analysis

`lsfit()`

`lm()`

R offers the functions `lsfit()` (**l**east **s**quares **fit**) and `lm()` (**l**inear **m**odels, a more general function) for regression analysis. This section focuses on `lm()`, since it offers more "features", especially when it comes to testing significance of the coefficients.
Consider again the `beer` data. Assume that we want to fit the following model (a parabola) to the logs of beer: $\log(X_t) = \alpha_0 + \alpha_1 \cdot t + \alpha_2 \cdot t^2 + e_t$
The fitting can be carried out in **R** with the following commands:

```
lbeer <- log(beer)
t <- seq(1956, 1995 + 7/12, length = length(lbeer))
t2 <- t^2
plot(lbeer)
lm(lbeer~t+t2)
lines(lm(lbeer~t+t2)$fit, col = 2, lwd = 2)
```

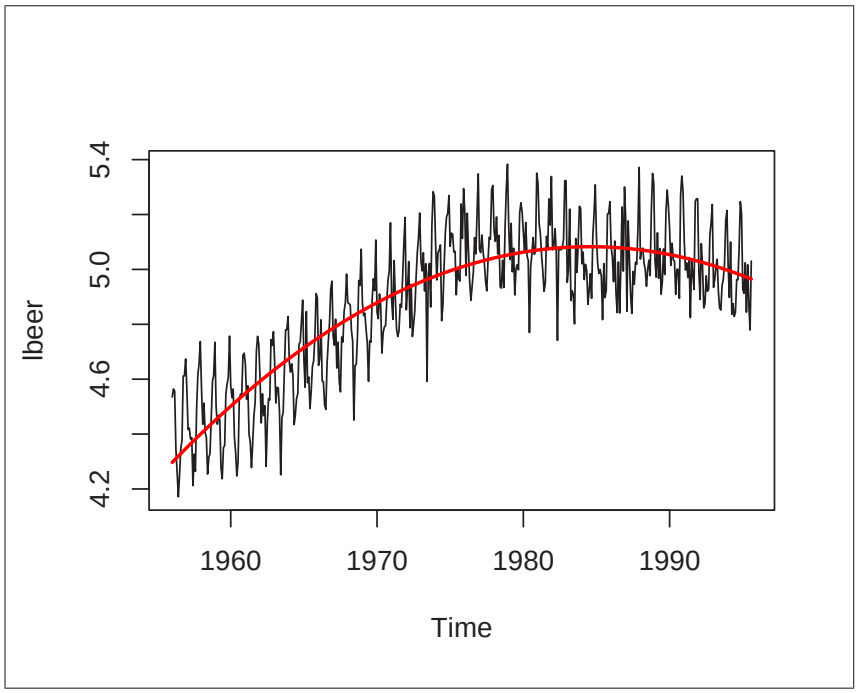

Figure 3.3: *Fitting a parabola to* `lbeer` *with* `lm()`.

In the first command above, logs of `beer` are computed and stored as `lbeer`. Explanatory variables (t and t^2 as `t` and `t2`) are defined in the second and third row. The actual fit of the model is done using `lm(lbeer~t+t2)`. The function `lm()` returns a `list`– object, whose element can be accessed using the "`$`"–sign: `lm(lbier~t+t2)$coefficients` returns the estimated coefficients (α_0, α_1 and α_2); `lm(lbier~t+t2)$fit` returns the fitted values $\hat{X}_t$ of the model. Extending the model to

$$\log(X_t) = \alpha_0 + \alpha_1 \cdot t + \alpha_2 \cdot t^2 + \beta \cdot \cos\left(\frac{2\pi t}{12}\right) + \gamma \cdot \sin\left(\frac{2\pi t}{12}\right) + e_t$$

so that it includes the first Fourier frequency is straightforward. After defining the two additional explanatory variables, `cos.t` and `sin.t`, the model can be estimated in the usual way:

```
lbeer <- log(beer)
t <- seq(1956, 1995 + 7/12, length = length(lbeer))
t2 <- t^2
sin.t <- sin(2*pi*t)
cos.t <- cos(2*pi*t)
plot(lbeer)
lines(lm(lbeer~t+t2+sin.t+cos.t)$fit, col = 4)
```

Note that in this case `sin.t` and `cos.t` do not include 12 in the denominator, since $\frac{1}{12}$ has already been considered during the transformation of `beer` and the

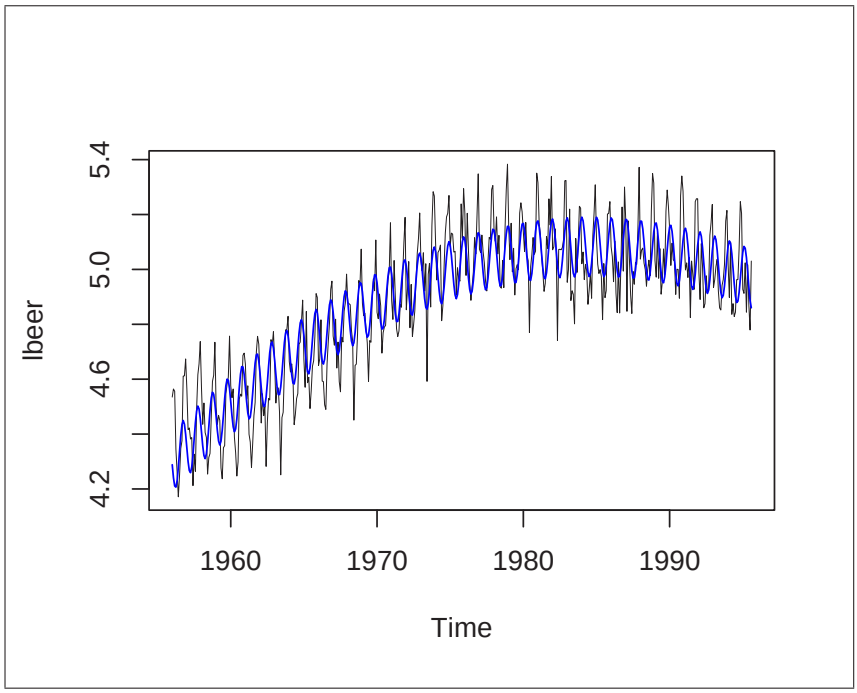

Figure 3.4: *Fitting a parabola and the first fourier frequency to* `lbeer`.

construction of `t`.

Another important aspect in regression analysis is to test the significance of the coefficients.
In the case of `lm()`, one can use the `summary()`– command: `summary()`

```
summary(lm(lbeer~t+t2+sin.t+cos.t))
```

which returns the following output:

```
Call:
lm(formula = lbeer ~ t + t2 + sin.t + cos.t)

Residuals:
       Min         1Q     Median         3Q        Max
-0.2722753 -0.0686953 -0.0006432  0.0695916  0.2370383

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -3.734e+03  1.474e+02 -25.330  < 2e-16 ***
t            3.768e+00  1.492e-01  25.250  < 2e-16 ***
t2          -9.493e-04  3.777e-05 -25.137  < 2e-16 ***
sin.t       -4.870e-02  6.297e-03  -7.735 6.34e-14 ***
cos.t        1.361e-01  6.283e-03  21.655  < 2e-16 ***
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.09702 on 471 degrees of freedom
Multiple R-Squared: 0.8668,     Adjusted R-squared: 0.8657
F-statistic: 766.1 on 4 and 471 DF,  p-value: < 2.2e-16
```

Apart from the coefficient estimates and their standard error, the output also includes the corresponding t-statistics and p–values. In our case, the coefficients α_0 (Intercept), α_1 (t), α_2 (t^2) and β (sin(t)) differ significantly from zero, while γ does not seem to. (One might include γ anyway, since Fourier frequencies are usually taken in pairs of sine and cosine.)

3.2 Exponential Smoothing

Introductory Remarks

One method of forecasting the next value x_{n+1}, of a time series $x_t, t = 1, 2, \ldots, n$ is to use a weighted average of past observations:

$$\hat{x}_n(1) = \lambda_0 \cdot x_n + \lambda_1 \cdot x_{n-1} + \ldots$$

The popular method of exponential smoothing assigns geometrically decreasing weights:

$$\lambda_i = \alpha(1-\alpha)^i \qquad ; \quad 0 < \alpha < 1$$

such that $\hat{x}_n(1) = \alpha \cdot x_n + \alpha(1-\alpha) \cdot x_{n-1} + \alpha(1-\alpha)^2 \cdot x_{n-2} + \ldots$

In its basic form exponential smoothing is applicable to time series with no systematic trend and/or seasonal components. It has been generalized to the "Holt–Winters"–procedure in order to deal with time series containing trend and seasonal variation. In this case, three smoothing parameters are required, namely α (for the level), β (for the trend) and γ (for the seasonal variation).

Exponential Smoothing and Prediction of Time Series

`HoltWinters()`

The `ts` – package offers the function `HoltWinters(x,alpha,beta,gamma)`, which lets one apply the Holt–Winters procedure to a time series `x`. One can specify the three smoothing parameters with the options `alpha`, `beta` and `gamma`. Particular components can be excluded by setting the value of the corresponding parameter to zero, e.g. one can exclude the seasonal component by specifying `gamma=0`. If one does not specify smoothing parameters, these are computed "automatically" (i.e. by minimizing the mean squared prediction error from the one–step–ahead forecasts).

Thus, the exponential smoothing of the `beer` dataset can be performed as follows:

```
beer <- read.csv("C:/beer.csv", header = TRUE, dec = ",",
                 sep = ";")
beer <- ts(beer[,1], start = 1956, freq = 12)
```

The above commands load the dataset from the CSV–file and transform it to a `ts`– object.

```
HoltWinters(beer)
```

This performs the Holt–Winters procedure on the `beer` dataset. It displays a list with e.g. the smoothing parameters ($\alpha \approx 0.076$, $\beta \approx 0.07$ and $\gamma \approx 0.145$ in this case). Another component of the list is the entry `fitted`, which can be accessed using `HoltWinters(beer)$fitted`:

```
plot(beer)
lines(HoltWinters(beer)$fitted[,1], col = "red")
```

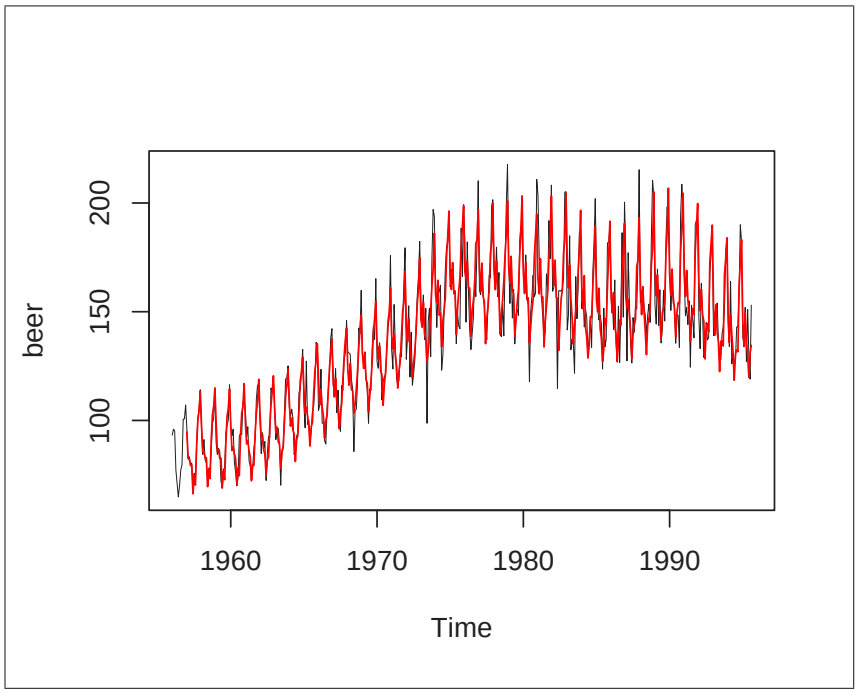

Figure 3.5: *Exponential smoothing of the* `beer` *data.*

R offers the function `predict()`, which is a generic function for predictions from various models. In order to use `predict()`, one has to save the "fit" of a model to an object, e.g.:

```
beer.hw <- HoltWinters(beer)
```

In this case, we have saved the "fit" from the Holt–Winters procedure on `beer` as `beer.hw`.

```
predict(beer.hw, n.ahead = 12)
```

predict()

returns the predicted values for the next 12 periods (i.e. Sep. 1995 to Aug. 1996). The following commands can be used to create a graph with the predictions for the next 4 years (i.e. 48 months):

```
plot(beer, xlim=c(1956, 1999))
lines(predict(beer.hw, n.ahead = 48), col = 2)
```

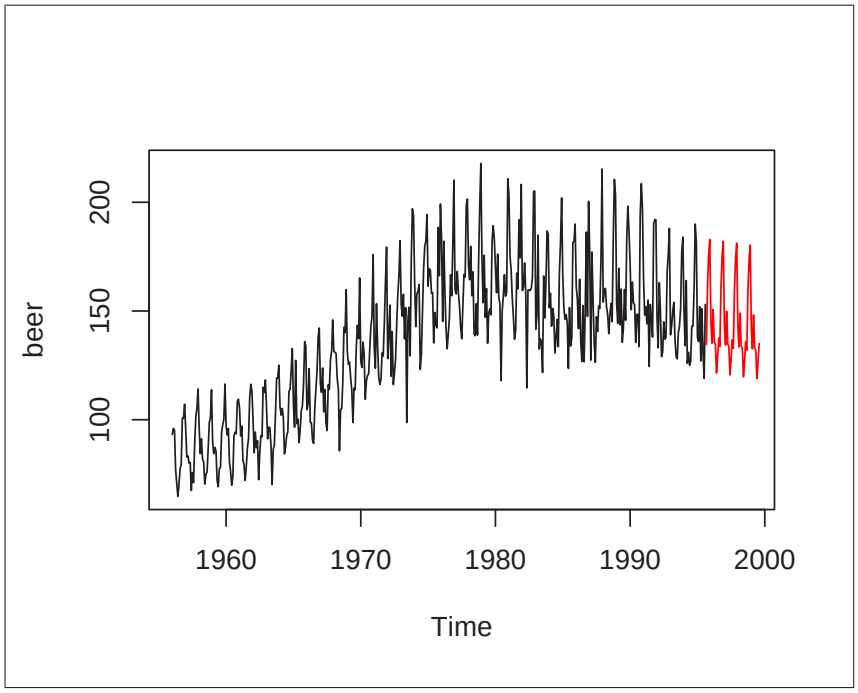

Figure 3.6: *Predicting* `beer` *with exponential smoothing.*

3.3 ARIMA–Models

Introductory Remarks

Forecasting based on ARIMA (autoregressive integrated moving average) models, sometimes referred to as the Box–Jenkins approach, comprises following stages:

i.) *Model identification*

ii.) *Parameter estimation*

iii.) *Diagnostic checking*

These stages are repeated iteratively until a satisfactory model for the given data has been identified (e.g. for prediction). The following three sections show some facilities that **R** offers for carrying out these three stages.

Analysis of Autocorrelations and Partial Autocorrelations

A first step in analysing time series is to examine the autocorrelations (ACF) and partial autocorrelations (PACF). **R** provides the functions `acf()` and `pacf()` for computing and plotting of ACF and PACF. The order of "pure" AR and MA processes can be identified from the ACF and PACF as shown below:

`acf()`

`pacf()`

`arima.sim()`

```
sim.ar <- arima.sim(list(ar = c(0.4, 0.4)), n = 1000)
sim.ma <- arima.sim(list(ma = c(0.6, -0.4)), n = 1000)
par(mfrow = c(2, 2))
acf(sim.ar, main = "ACF of AR(2) process")
acf(sim.ma, main = "ACF of MA(2) process")
pacf(sim.ar, main = "PACF of AR(2) process")
pacf(sim.ma, main = "PACF of MA(2) process")
```

The function `arima.sim()` was used to simulate the ARIMA(p,d,q)–models:

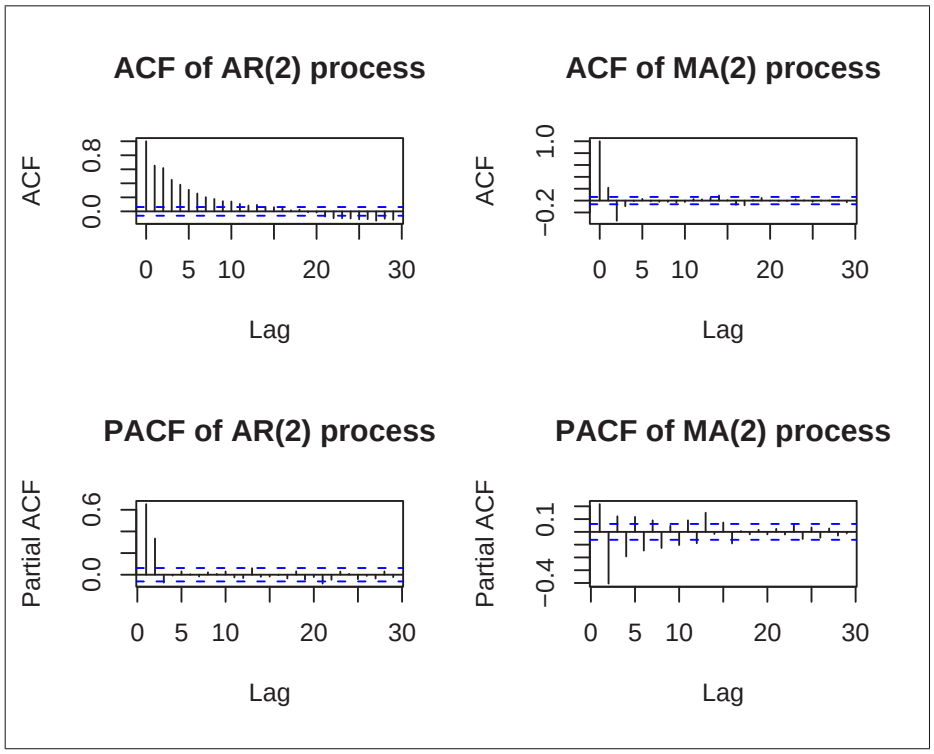

Figure 3.7: *ACF and PACF of AR– and MA–models.*

In the first line 1000 observations of an ARIMA(2,0,0)–model (i.e. AR(2)–model) were simulated and saved as `sim.ar`. Equivalently, the second line simulated 1000 observations from a MA(2)–model and saved them to `sim.ma`.
An useful command for graphical displays is `par(mfrow=c(h,v))` which splits the graphics window into (h×v) regions — in this case we have set up 4 separate regions within the graphics window.

The last four lines create the ACF and PACF plots of the two simulated processes. Note that by default the plots include confidence intervals (based on uncorrelated series).

Estimating Parameters of ARIMA–Models

Once the order of the ARIMA(p,d,q)–model has been specified, the parameters can be estimated using the function `arima()` from the `ts`–package: `arima()`

```
arima(data, order = c(p, d, q))
```

Fitting e.g. an ARIMA(1,0,1)–model on the `LakeHuron`–dataset (annual levels of the Lake Huron from 1875 to 1972) is done using

`data()`

```
data(LakeHuron)
fit <- arima(LakeHuron, order = c(1, 0, 1))
```

In this case `fit` is a list containing e.g. the coefficients (`fit$coef`), residuals (`fit$residuals`) and the Akaike Information Criterion AIC (`fit$aic`).

Diagnostic Checking

A first step in diagnostic checking of fitted models is to analyse the residuals from the fit for any signs of non–randomness. **R** has the function `tsdiag()`, which produces a diagnostic plot of a fitted time series model: `tsdiag()`

```
fit <- arima(LakeHuron, order = c(1, 0, 1))
tsdiag(fit)
```

It produces the output shown in figure 3.8: A plot of the residuals, the autocorrelation of the residuals and the p-values of the Ljung–Box statistic for the first 10 lags.
The Box–Pierce (and Ljung–Box) test examines the Null of independently distributed residuals. It's derived from the idea that the residuals of a "correctly specified" model are independently distributed. If the residuals are not, then they come from a miss–specified model. The function `Box.test()` computes the test statistic for a given lag: `Box.test()`

```
Box.test(fit$residuals, lag = 1)
```

Prediction of ARIMA–Models

Once a model has been identified and its parameters have been estimated, one can predict future values of a time series. Let's assume that we are satisfied with

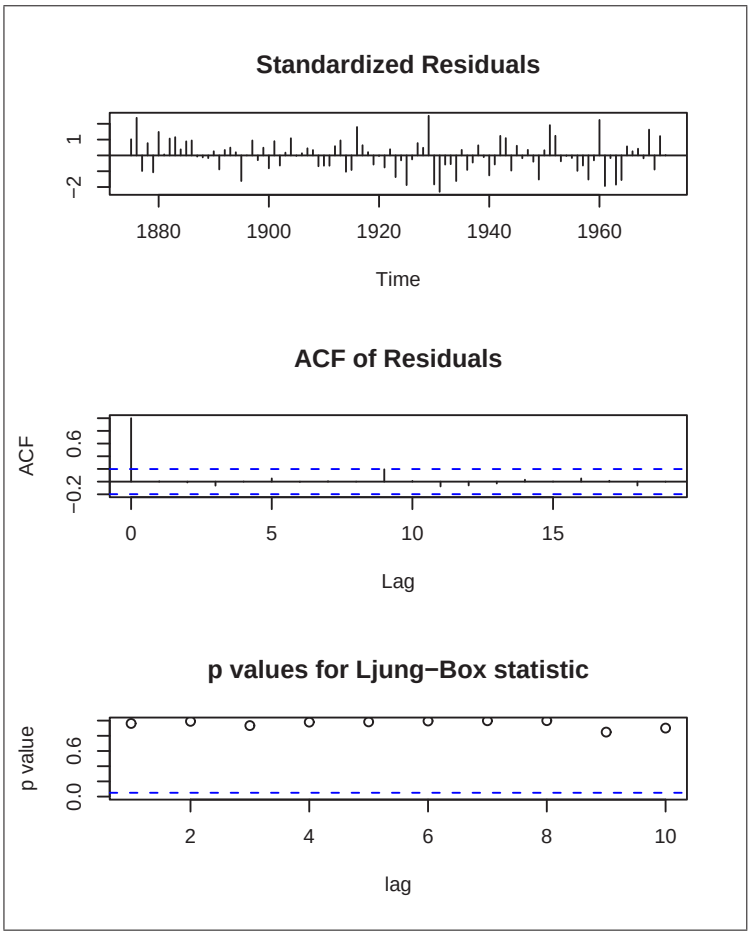

Figure 3.8: *Output from* `tsdiag()`.

the fit of an ARIMA(1,0,1)–model to the `LakeHuron`–data:

```
fit <- arima(LakeHuron, order = c(1, 0, 1))
```

As with Exponential Smoothing, the function `predict()` can be used for predicting future values of the levels under the model:

`predict()`

```
LH.pred <- predict(fit, n.ahead = 8)
```

Here we have predicted the levels of Lake Huron for the next 8 years (i.e. until 1980). In this case, `LH.pred` is a list containing two entries, the predicted values `LH.pred$pred` and the standard errors of the prediction `LH.pred$se`. Using the familiar rule of thumb for an approximate confidence interval (95%) for the prediction, *"prediction* $\pm$ *2·SE"*, one can plot the Lake Huron data, the predicted values and the corresponding approximate confidence intervals:

```
plot(LakeHuron, xlim = c(1875, 1980), ylim = c(575, 584))
LH.pred <- predict(fit, n.ahead = 8)
```

```
lines(LH.pred$pred, col = "red")
lines(LH.pred$pred + 2*LH.pred$se, col = "red", lty = 3)
lines(LH.pred$pred - 2*LH.pred$se, col = "red", lty = 3)
```

First, the levels of Lake Huron are plotted. In order to leave some space for adding the predicted values, the x-axis has been set to the interval 1875 to 1980 using the optional argument `xlim=c(1875,1980)`; the use of `ylim` below is purely for visual enhancement. The prediction takes place in the second line using `predict()` on the fitted model. Adding the prediction and the approximate confidence interval is done in the last three lines. The confidence bands are drawn as a red, dotted line (using the options `col="red"` and `lty=3`):

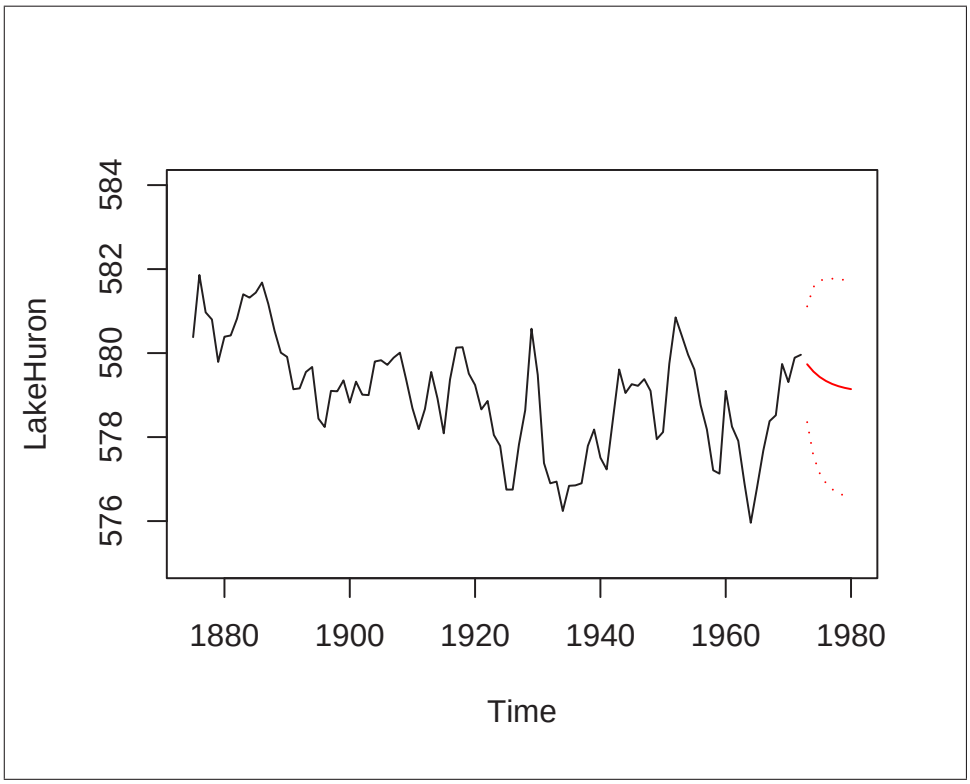

Figure 3.9: *Lake Huron levels and predicted values.*

Chapter 4

Advanced Graphics

4.1 Customizing Plots

Labelling graphs
R offers various means for annotating graphs. Consider a histogram of 100 normally distributed random numbers given by

```
hist(rnorm(100), prob = TRUE)
```

Assume that we want to have a custom title and different labels for the axes as shown in figure 4.1. The relevant options are `main` (for the title), `xlab` and `ylab` (axes labels):

`main`
`xlab`
`ylab`

```
hist(rnorm(100), prob = TRUE, main = "custom title",
     xlab = "x label", ylab = "y label")
```

The title and the labels are entered as characters, i.e. in quotation marks. To include quotation marks in the title itself, a backslash is required before each quotation mark: `\"`. The backslash is also used for some other commands, such as line breaks. Using `\n` results in a line feed, e.g.

```
main = "first part \n second part"
```

within a plot command writes *"first part"* in the first line of the title and *"second part"* in the second line.

Setting font face and font size
The option `font` allows for (limited) control over the font type used for annotations. It is specified by an integer. Additionally, different font types can be used for different parts of the graph:

`font`

- `font.axis` *# specifies the font for the axis annotations* — `font.axis`
- `font.lab` *# specifies the font for axis labels* — `font.lab`

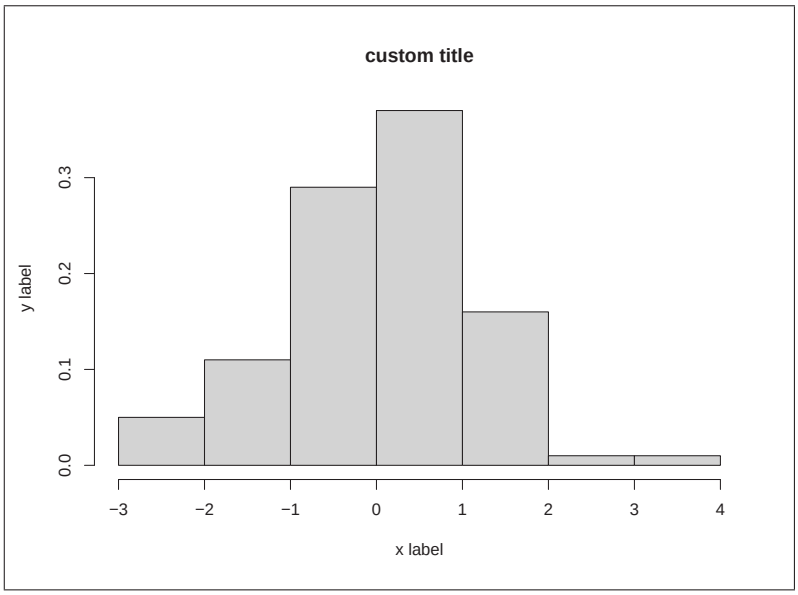

Figure 4.1: *Customizing the main title and the axes labels using* `main`*,* `xlab` *and* `ylab`*.*

- `font.main` *# specifies the font for the (main) title* **font.main**
- `font.sub` *# specifies the font for the subtitle* **font.sub**

The use of the `font`–options is illustrated in the example below:

```
hist(rnorm(100), sub = "subtitle", font.main = 6,
     font.lab = 7, font.axis = 8, font.sub = 9)
```

sub

Integer codes used in this example are:

- `6` : "Times" font
- `7` : "Times" font, italic
- `8` : "Times" font, boldface
- `9` : "Times" font, italic and boldface

The text size can be controlled using the `cex` option ("character expansion"). Again, the `cex` option also has "subcategories" such as `cex.axis` , `cex.lab` , `cex.main` and `cex.sub` . The size of text is specified using a relative value (e.g. `cex=1` doesn't change the size, `cex=0.8` reduces the size to 80% and `cex=1.2` enlarges the size to 120%).
A complete list of "graphical parameters" is given in the help–file for the `par()`–command, i.e. by typing

cex
cex.axis
cex.lab
cex.main
cex.sub

```
?par
```
par()

into the console.
Another useful command for labelling graphs is `text(a, b, "content")`, **text()**
which adds *"content"* to an existing plot at the given coordinates $(x = \texttt{a}, y = \texttt{b})$:

```
hist(rnorm(500), prob = TRUE)
text(2, 0.2, "your text here")
```

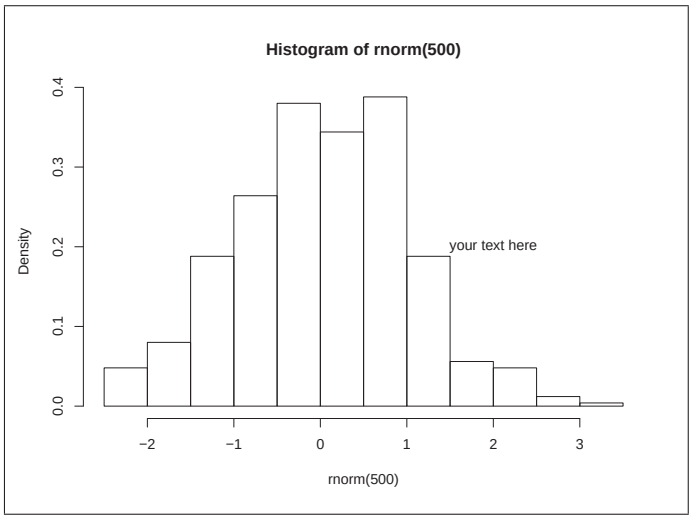

Figure 4.2: *Adding text to an existing plot using* `text()`.

Specification of Colours
There exist various means of specifying colours in **R**. One way is to use the "**R** names" such as `col="blue"`, `col="red"` or even `col="mediumslateblue"`. **col**
(A complete list of available colour names is obtained with `colours()`.) Alter- **colours()**
natively, one can use numerical codes to specify the colours, e.g. `col=2` (for red), `col=3` (for green) etc. Colours can also be specified in hexadecimal code (as in html), e.g. `col="#FF0000"` denotes to red. Similarly, one can use `col=rgb(1,0,0)` **rgb()**
for red. The `rgb()` command is especially useful for custom colour palettes.
R offers a few predefined colour palettes. These are illustrated on the `volcano`–data example below:

```
data(volcano)
par(mfrow = c(2, 2))
image(volcano, main = "heat.colors")
image(volcano, main = "rainbow", col = rainbow(15))
image(volcano, main = "topo", col = topo.colors(15))
image(volcano, main = "terrain.colors",
      col = terrain.colors(15))
```
data()

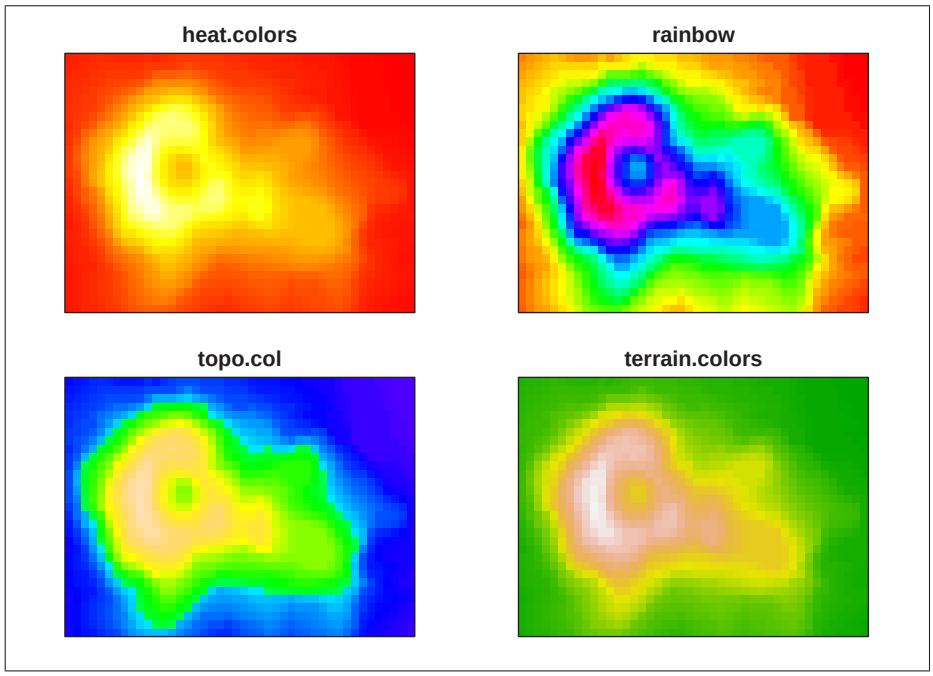

Figure 4.3: *Some predefined colour palettes available in* **R**.

The resulting image maps with different colour palettes are shown in figure 4.3. The (internal) dataset "volcano", containing topographic information for Maunga Whau on a 10m by 10m grid, is loaded by entering `data(volcano)`. The command `par(mfrow=c(a,b))` is used to split the graphics window into $a \cdot b$ regions (a "rows" and b "columns"). The `image()` function creates an image map of a given matrix.

`image()`

4.2 Mathematical Annotations

Occasionally, it is useful to add mathematical annotations to plots. Let's assume we want to investigate the relationship between `HP` (horsepower) and `MPG` (miles per gallon) from the `car` dataset by fitting the following two models to the data

$$M_1\text{: } MPG_i = \beta_0 + \beta_1 \cdot HP_i + e_i$$

$$M_2\text{: } MPG_i = \beta_0 + \beta_1 \cdot HP_i + \beta_2 \cdot HP_i^2 + e_i$$

Fitting the model and plotting the observations along with the two fitted models is done with

```
car <- read.table("C:/R_workshop/car.dat", header = TRUE)
 attach(car)
M1 <- lm(MPG~HP)
HP2 <- HP^2
```

```
M2 <- lm(MPG~HP+HP2)
plot(HP, MPG, pch = 16)
x <- seq(0, 350, length = 500)
y1 <- M1$coef[1] + M1$coef[2]*x
y2 <- M2$coef[1] + M2$coef[2]*x + M2$coef[3]*x^2
lines(x, y1, col="red")
lines(x, y2, col="blue")
```

In order to add mathematical expressions, **R** offers the function `expression()` which can be used e.g. in conjunction with the `text` command:

expression()

```
text(200, 55, expression(bold(M[1])*":"*hat(MPG)==
     hat(beta)[0] + hat(beta)[1]*"HP"),
     col = "red", adj = 0)

text(200, 50, expression(bold(M[2])*":"*hat(MPG)==
     hat(beta)[0] + hat(beta)[1]*"HP" + hat(beta)[2]*"HP"^2),
     col = "blue", adj = 0)
```

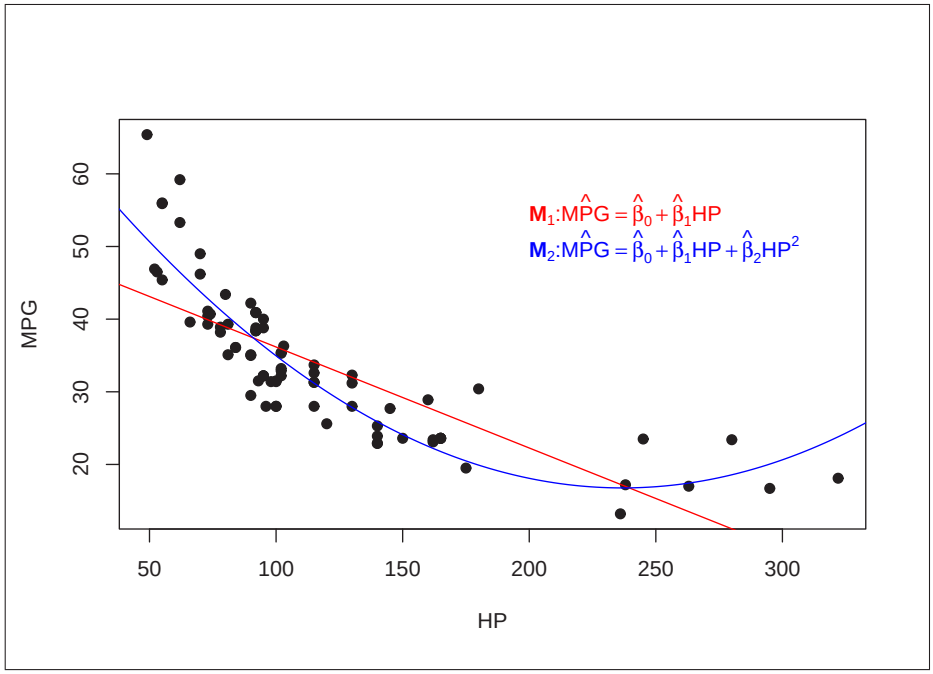

Figure 4.4: *Using* `expression()` *for mathematical annotations.*

Further options for annotating plots can be found in the examples given in the help documentation of the `legend()` function. A list of available expressions is given in the appendix.

legend()

4.3 Three-Dimensional Plots

Perspective plots

The **R** function `persp()` can be used to create 3D plots of surfaces. A 3D display of the `volcano`– data can be created with

`persp()`

```
data(volcano)
persp(volcano)
persp(volcano, theta = 70, phi = 40)
```

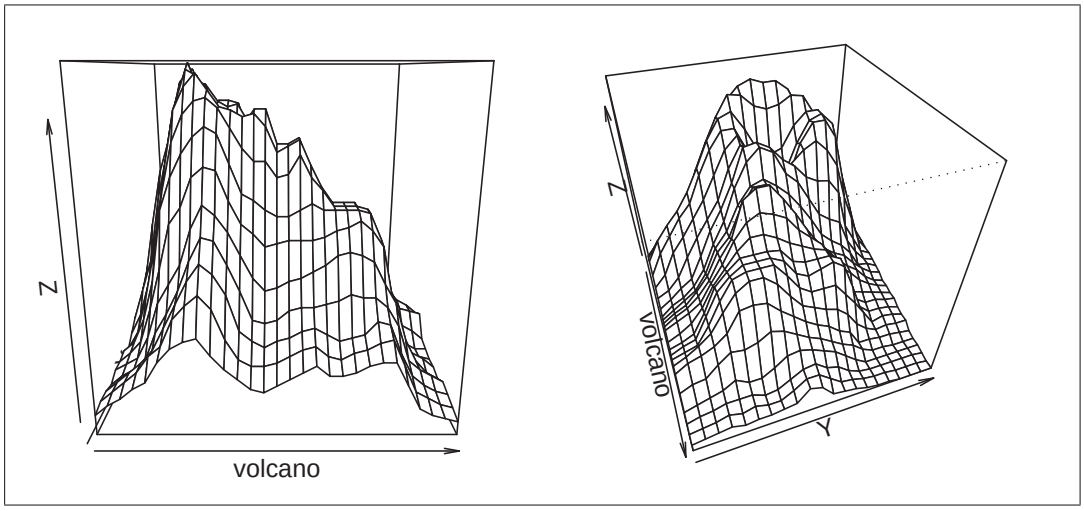

Figure 4.5: *3D plots with* `persp()`.

The 3D space can be "navigated" by changing the parameters `theta` (azimuthal direction) and `phi` (colatitude).

`theta`

`phi`

Further options are illustrated in the example below:

```
par(mfrow=c(1,2))

# example 1:
persp(volcano, col = "green", border = NA, shade = 0.9,
      theta = 70, phi = 40, ltheta = 120, box = FALSE,
      axes = FALSE, expand = 0.5)

# example 2:
collut <- terrain.colors(101)
temp <- 1 + 100*(volcano-min(volcano)) /
        (diff(range(volcano)))
mapcol <- collut[temp[1:86, 1:61]]
persp(volcano, col = mapcol, border = NA, theta = 70,
      phi = 40,shade = 0.9, expand = 0.5, ltheta = 120,
```

```
        lphi = 30)
```

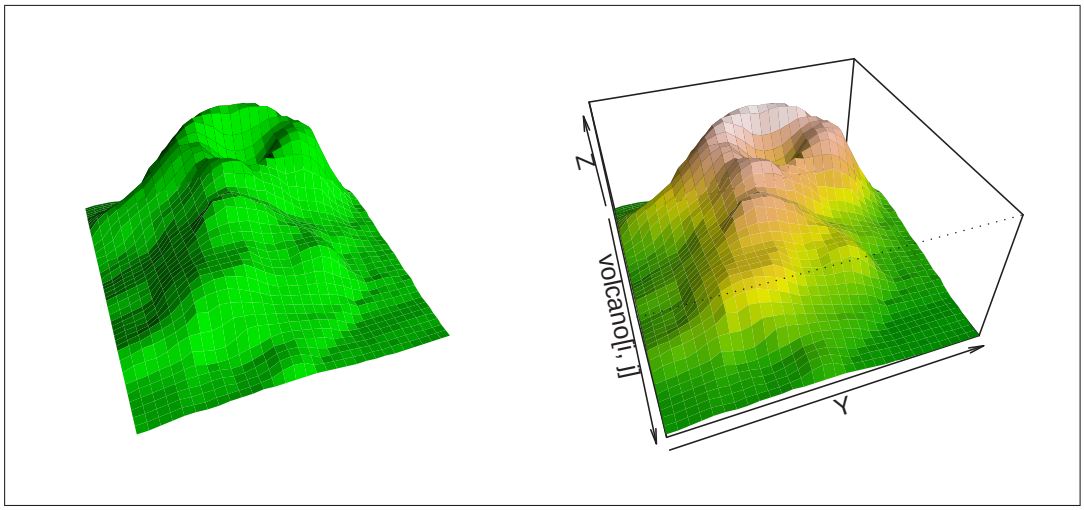

Figure 4.6: *Advanced options for* `persp()`.

Plotting functions of two variables

In order to display two–dimensional functions $f(x, y)$ with `persp()` the following **R** objects are required: x, y (grid mark vectors) and the values $z = f(x, y)$ which are stored as a matrix. A useful function here is `outer(x,y,f)`, which computes the values of $z = f(x, y)$ for all pairs of entries in the vectors x and y. In the example given below the vectors x and y are specified and then the function `f` is defined. The command `outer()` creates a matrix, which is stored as `z`. It contains the values of $f(x, y)$ for all points on the specified x–y grid. Finally, `persp()` is used to generate the plot (figure 4.7):

`outer()`

```
y <- x <- seq(-2, 2, length = 20)
f <- function(x, y)
  {
  fxy <- -x^2 - y^2
  return(fxy)
  }
z <- outer(x, y, f)
persp(x, y, z, theta = 30, phi = 30)
```

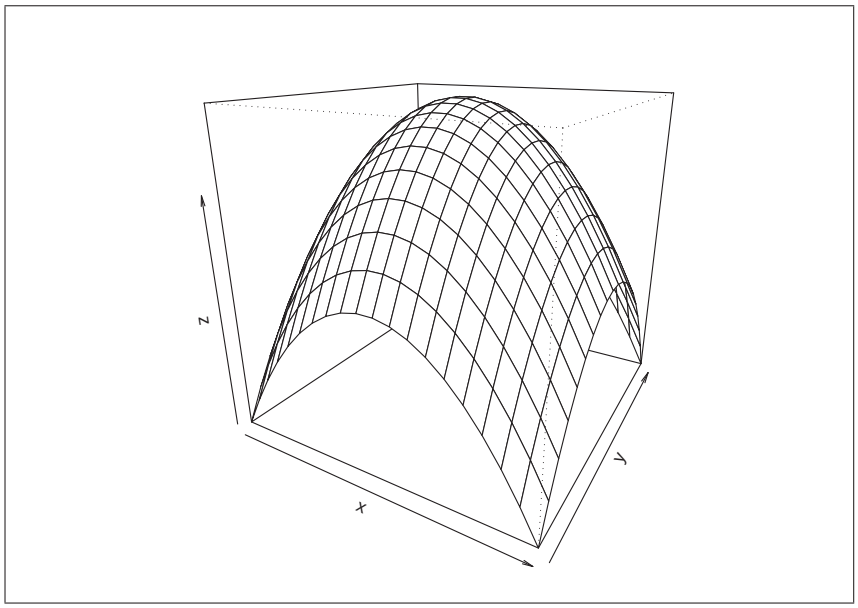

Figure 4.7: *Plotting 2D functions with* `persp()`.

4.4 RGL: 3D Visualization in R using OpenGL

RGL is an **R** package which was designed to overcome some limitations for 3D graphics. It uses OpenGL © as the rendering backend and is freely available at the URI

http://134.76.173.220/~dadler/rgl/index.html

Further information on RGL can be found the website and on the slides "RGL: An **R**-Library for 3D Visualization in R" *('rgl.ppt')* in your working directory.

Appendix A

R–functions

A.1 Mathematical Expressions (`expression()`)

• ARITHMETIC OPERATORS:

Expression	*Result*
`x+y`	$\mathbf{x + y}$
`x-y`	$\mathbf{x - y}$
`x*y`	$\mathbf{xy}$
`x/y`	$\mathbf{x/y}$
`x%+-%y`	$\mathbf{x \pm y}$
`x%/%y`	$\mathbf{x \div y}$
`x%*%y`	$\mathbf{x \times y}$
`-x`	$\mathbf{-x}$
`+x`	$\mathbf{+x}$

• SUB- AND SUPERSCRIPTS:

Expression	*Result*
`x[i]`	$\mathbf{x_i}$
`x^2`	$\mathbf{x^2}$

• JUXTAPOSITION:

Expression	*Result*
`x*y`	$\mathbf{xy}$
`paste(x,y,z)`	$\mathbf{xyz}$

• LISTS:

Expression	*Result*
`list(x,y,z)`	$\mathbf{x, y, z}$

• RADICALS:

Expression	*Result*
`sqrt(x)`	$\sqrt{\mathbf{x}}$
`sqrt(x,y)`	$\sqrt[y]{x}$

• RELATIONS:

Expression	*Result*
`x==y`	$\mathbf{x = y}$
`x!=y`	$\mathbf{x \neq y}$
`x<y`	$\mathbf{x < y}$
`x<=y`	$\mathbf{x \leq y}$
`x>y`	$\mathbf{x > y}$
`x>=y`	$\mathbf{x \geq y}$
`x%~~%y`	$\mathbf{x \approx y}$
`x%~=%y`	$\mathbf{x \cong y}$
`x%==%y`	$\mathbf{x \equiv y}$
`x%prop%y`	$\mathbf{x \propto y}$

• SYMBOLIC NAMES:

Expression	*Result*
`Alpha-Omega`	$\mathbf{A - \Omega}$
`alpha-omega`	$\alpha - \omega$
`infinity`	∞
`32*degree`	32°
`60*minute`	$32'$
`30*second`	$32''$

• ELLIPSIS:

Expression	*Result*
`list(x[1],...,x[n])`	$\mathbf{x_1, \ldots, x_n}$
`x[1]+...+x[n]`	$\mathbf{x_1 + \cdots + x_n}$
`list(x[1],cdots,x[n])`	$\mathbf{x_1, \cdots, x_n}$
`x[1]+ldots+x[n]`	$\mathbf{x_1 + \ldots + x_n}$

• SET RELATIONS:

Expression	*Result*
`x%subset%y`	$x \subset y$
`x%subseteq%y`	$x \subseteq y$
`x%supset%y`	$x \supset y$
`x%supseteq%y`	$x \supseteq y$
`x%notsubset%y`	$x \not\subset y$
`x%in%y`	$x \in y$
`x%notin%y`	$x \notin y$

• ACCENTS:

Expression	*Result*
`hat(x)`	$\hat{x}$
`tilde(x)`	$\tilde{x}$
`ring(x)`	$\mathring{x}$
`bar(x)`	$\bar{x}$
`widehat(xy)`	$\widehat{xy}$
`widetilde`	$\widetilde{xy}$

• ARROWS:

Expression	*Result*
`x%<->%y`	$x \leftrightarrow y$
`x%->%y`	$x \rightarrow y$
`x%<-%y`	$x \leftarrow y$
`x%up%y`	$x \uparrow y$
`x%down%y`	$x \downarrow y$
`x%<=>%y`	$x \Leftrightarrow y$
`x%=>%y`	$x \Rightarrow y$
`x%<=%y`	$x \Leftarrow y$
`x%dblup%y`	$x \Uparrow y$
`x%dbldown%y`	$x \Downarrow y$

• SPACING:

Expression	*Result*
`x~ ~y`	$x\ y$
`x+phantom(0)+y`	$x + + y$
`x+over(1,phantom(0))`	$x + \frac{1}{}$

• FRACTIONS:

Expression	*Result*
`frac(x,y)`	$\frac{x}{y}$
`over(x,y)`	$\frac{x}{y}$
`atop(x,y)`	$\begin{matrix} x \\ y \end{matrix}$

• STYLE:

Expression	*Result*
`displaystyle(x)`	$\displaystyle x$
`textstyle(x)`	$\textstyle x$
`scriptstyle(x)`	$\scriptstyle x$
`scriptscriptstyle(x)`	$\scriptscriptstyle x$

• TYPEFACE:

Expression	*Result*
`plain(x)`	x
`italic(x)`	x
`bold(x)`	$\mathbf{x}$
`bolditalic(x)`	$\boldsymbol{x}$

• BIG OPERATORS:

Expression	*Result*
`sum(x[i],i=1,n)`	$\sum_{1}^{n} x_i$
`prod(plain(P)(X==x),x)`	$\prod_{x} P(X=x)$
`integral(f(x)*dx,a,b)`	$\int_{a}^{b} f(x)dx$
`union(A[i],i==1,n)`	$\bigcup_{i=1}^{n} A_i$
`intersect(A[i],i==1,n)`	$\bigcap_{i=1}^{n} A_i$
`lim(f(x),x%->%0)`	$\lim_{x \rightarrow 0} f(x)$
`min(g(x),x>=0)`	$\min_{x \geq 0} g(x)$
`inf(S)`	$\inf S$
`sup(S)`	$\sup S$

• GROUPING:

Expression	*Result*
`(x+y)*z`	$(x+y)z$
`x^y+z`	$x^{y}+z$
`x^(y+z)`	$x^{(y+z)}$
`x^{y+z}`	x^{y+z}
`group("(",list(a,b),"]")`	$(a,b]$
`bgroup("(",atop(x,y),")")`	$\binom{x}{y}$
`group(lceil,x,rceil)`	$\lceil x \rceil$
`group(lfloor,x,rfloor)`	$\lfloor x \rfloor$
`group("|",x,"|")`	$\lvert x \rvert$

A.2 The RGL Functionset

• DEVICE MANAGEMENT:	
`rgl.open()`	Opens a new device.
`rgl.close()`	Closes the current device.
`rgl.cur()`	Returns the number of the active device.
`rgl.set(which)`	Sets a device as active.
`rgl.quit()`	Shuts down the subsystem and detaches RGL.
• SCENE MANAGEMENT:	
`rgl.clear(type="shapes")`	Clears the scene from the stack of specified type ("shapes" or "lights").
`rgl.pop(type="shapes")`	Removes the last added node from stack.
• EXPORT FUNCTIONS:	
`rgl.snapshot(file)`	Saves a screenshot of the current scene in PNG–format.
• SHAPE FUNCTIONS:	
`rgl.points(x,y,z,...)`	Add points at (x, y, z).
`rgl.lines(x,y,z,...)`	Add lines with nodes (x_i, y_i, z_i), $i = 1, 2$.
`rgl.triangles(x,y,z,...)`	Add triangles with nodes (x_i, y_i, z_i), $i = 1, 2, 3$.
`rgl.quads(x,y,z,...)`	Add quads with nodes (x_i, y_i, z_i), $i = 1, 2, 3, 4$.
`rgl.spheres(x,y,z,r,...)`	Add spheres with center (x, y, z) and radius r.
`rgl.texts(x,y,z,text,...)`	Add texts at (x, y, z).
`rgl.sprites(x,y,z,r,...)`	Add 3D sprites at (x, y, z) and half-size r.
`rgl.surface(x,y,z,...)`	Add surface defined by two grid mark vectors x and y and a surface height matrix z.
• ENVIRONMENT SETUP:	
`rgl.viewpoint(theta,phi, fov,zoom,interactive)`	Sets the viewpoint (`theta`, `phi`) in polar coordinates with a field–of–view angle `fov` and a zoom factor `zoom`. The logical flag `interactive` specifies whether or not navigation is allowed.
`rgl.light(theta,phi,...)`	Adds a light source to the scene.
`rgl.bg(...)`	Sets the background.
`rgl.bbox(...)`	Sets the bounding box.
• APPEARANCE FUNCTIONS:	
`rgl.material(...)`	Generalized interface for appearance parameters.