Time Series Analysis with R - Part I

Walter Zucchini, Oleg Nenadić

Contents

Chapter 1

Getting started

1.1 Downloading and Installing R

R is a widely used environment for statistical analysis. The striking difference between **R** and most other statistical software is that it is free software and that it is maintained by scientists for scientists. Since its introduction in 1996, the R–project has gained many users and contributors, which continously extend the capabilities of **R** by releasing add–ons (packages) that offer previously not available 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 burden hab been taken, **R** offers almost unlimited possibilities for statistical data analysis.

R is distributed by the "Comprehensive R Archive Network" (CRAN) – it is available from the url: *http://cran.r-project.org*. The current version of **R** (1.7.0, approx. 20 MB) for Windows can be downloaded by selecting "*R binaries*" → "*windows*" → "*base*" and downloading the file "*rw1070.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 Data Preparation and Import in R

Importing data into **R** can be carried out in various ways – to name a few, **R** offers means for importing ASCII and binary data, data from other applications or even for database–connections. Since a common denominator for "data analysis" (*cough*) seem to be spreadsheet applications like e.g. Microsoft Excel ©, the

remainder of this section will focus on importing data from Excel to **R**.
Let's assume we have the following dataset `tui` as a spreadsheet in Excel. (The dataset can be downloaded from *http://134.76.173.220/tui.zip* as a zipped Excel–file).
The spreadsheet 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).
A convenient way of preparing the data is to clean up the table within Excel, so that only the data itself and one row containing the column–names remain.
Once the data has this form, it can be exported as a *CSV* ("comma seperated values") file, e.g. to `C:/tui.csv`.
After conversion to a CSV–file, the data can be loaded into **R**. In our case, the `tui`–dataset is imported by typing

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

into the **R**–console. The right–hand side, `read.csv( )`, is the **R**–command, which reads the CSV–file (`tui.csv`). Note, that paths use slashes "/" instead of backslashes. Further options within this command include `header` and `dec`, which specify if the dataset has the first row containing column–names (`header=T`). In case one does not have a naming row, one would use `header=F` instead. The option `dec` sets the decimal seperator used in case it differs from a point (here a comma is used as a decimal seperator).

1.3 Basic R–commands: Data Manipulation and Visualization

The dataset is stored as a matrix–object with the name `tui`. In order to access particular elements of objects, square brackets (`[ ]`) are used. Rows and colums of matrices can be accessed with `object[` *row*,*column*`]`.
The closing values of the TUI shares are stored in the 5th column, so they can be selected with `tui[,5]`. A chart with the closing values can be created using the `plot( )` – command:

```
plot(tui[,5],type="l")
```

The `plot( )` – command allows for a number of optional arguments, one of them is `type="l"`, which sets the plot–type to "lines". Especially graphics commands allow for a variety of additional options, e.g.:

```
plot(tui[,5], type="l",
  lwd=2, col="red", xlab="time", ylab="closing values",
      main="TUI AG", ylim=c(0,60) )
```

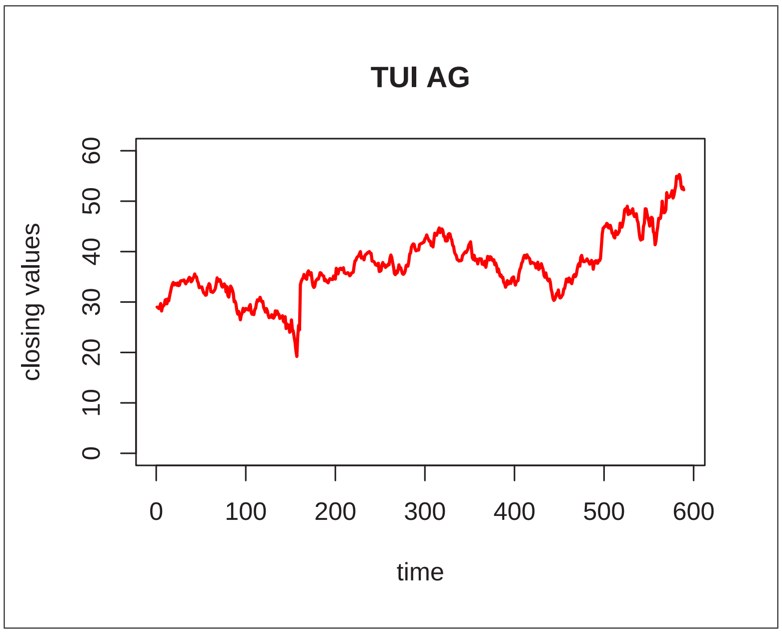

Figure 1.1: Creating charts with the `plot( )` – command

First it has to be noted that **R** allows commands to be longer than one line – the `plot( )` – command is evaluated after the closing bracket!
The option `lwd=2` (**l**ine **wid**th) is used to control the thickness of the plotted line and `col="red"` controls the colour used (a list of available colour–names for using within this option can be obtained by typing `colors()` into the console). `xlab` and `ylab` are used for labeling the axes while `main` specifies the title of the plot. Limits of the plotting region from **a** to **b** (for the x–and y–axes) can be set using the `xlim=c(`**a**`,`**b**`)` and/or `ylim=c(`**a**`,`**b**`)` options.
A complete list of available options can be displayed using the *help*– system of **R**. Typing `?plot` into the console opens the help–file for the `plot( )` – command (the helpfile for the graphical parameters is accessed with `?par`). Every **R**–function has a corresponding help–file which can be accessed by typing a question mark and the command (without brackets). It contains further details about the function and available options, references and examples of usage.
Now back to our TUI–shares. Assume that we want to plot differences of the logarithms of the returns. In order to do so, we need two operators, `log( )` which takes logarithms and `diff( )` which computes differences of a given object:

```
  plot(diff(log(tui[,5])),type="l")
```

Operations in **R** can be nested (`diff(log( ))`) as in the example above – one just needs to care about the brackets (each opening bracket requires a closing one)!
Another aspect to time series is to investigate distributional properties. A first step would be to draw a histogram and to compare it with e.g. the density of the normal distribution:

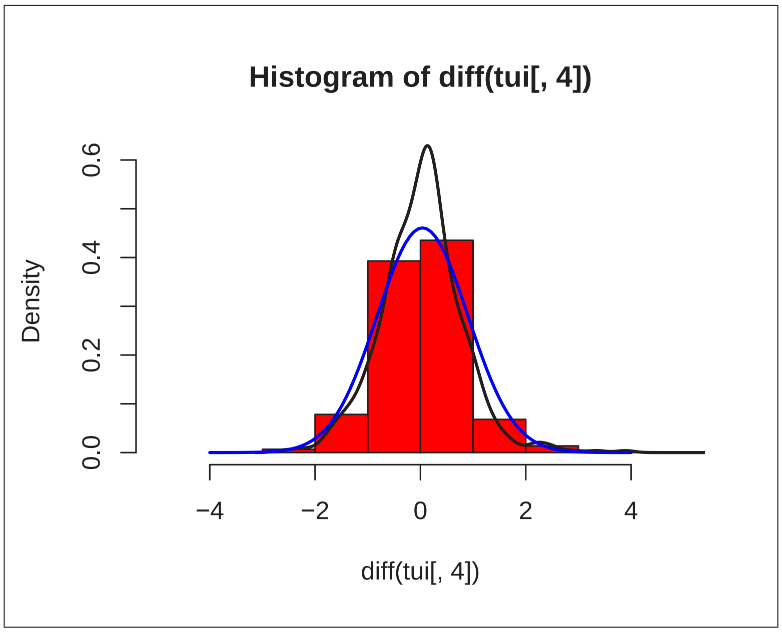

Figure 1.2: Comparing histograms with densities

The figure is created using following commands:

```
hist(diff(tui[,4]),prob=T,ylim=c(0,0.6),xlim=c(-5,5),col="red")
lines(density(diff(tui[,4])),lwd=2)
```

In the first line, the `hist( )` – command is used for creating a histogramm of the differences. The option `prob=T` causes the histogram to be displayed based on relative frequencies. The other options (`ylim`, `xlim` and `col`) are solely for enhancing the display. A nonparametric estimate of the density is added using the function `density( )` together with `lines( )`.
In order to add the density of the fitted normal distribution, one needs to estimate the mean and the standard deviation using `mean( )` and `sd( )`:

```
mu<-mean(diff(tui[,4]))
sigma<-sd(diff(tui[,4]))
```

Having estimated the mean and standard deviation, one additionally needs to define x–values for which the corresponding values of the normal distribution are to be calculated:

```
x<-seq(-4,4,length=100)
y<-dnorm(x,mu,sigma)
lines(x,y,lwd=2,col="blue")
```

`seq(a,b,length)` creates a sequence of `length` values from `a` to `b` – in this case 100 values from -4 to 4. The corresponding values of the normal distribution for given μ and σ are then computed using the function `dnorm(x,`μ`,`σ`)`. Adding "lines" to existing plots is done with the function `lines(x,y)` – the major difference to `plot(x,y)` is that `plot( )` clears the graphics–window.

Another option for comparison with the normal distribution is given using the function `qqnorm(diff(tui[,4]))` which draws a quantile- quantile plot:

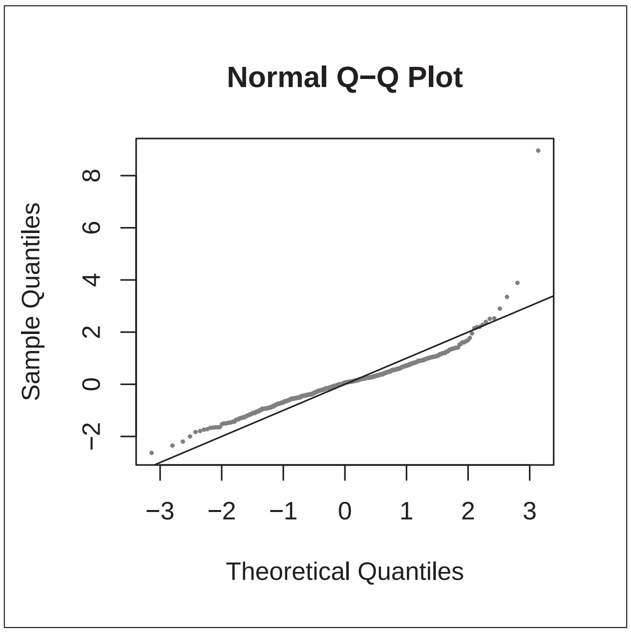

Figure 1.3: Comparing empirical with theoretical quantiles

Without going too much into detail, the ideal case (i.e. normally distributed observations) is given when the observations lie on the line (which is drawn using `abline(0,1)`).

There are various means for testing normality of given data. The Kolmogorov-Smirnoff test can be used to test if sample follows a specific distribution. In order to test the normality of the differences of the logs from the TUI-shares, one can use the function `ks.test( )` together with the corresponding distribution name:

```
x<-diff(log(tui[,5]))
ks.test(x,"pnorm",mean(x),sd(x))
```

Since we want to compare the data against the (fitted) normal distribution, we include `"pnorm"`, `mean(x)` and `sd(x)` (as result we should get a significant deviation from the normal distribution).
Unfortunately, the Kolmogorov–Smirnoff test does not take account of where the deviations come from – causes for that range from incorrectly estimated parameters, skewness to leptocurtic distributions. Another possibility (for testing normality) is to use the Shapiro–Test (which still does not take account of skewness etc., but generally performs a bit better):

```
shapiro.test(x)
```

Since this test is a test for normality, we do not have to specify a distribution but only the data which is to be tested (here the results should be effectively the same, except for a slightly different p-value).

Chapter 2

Simple Component Analysis

2.1 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 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 $\{\frac{1}{2a+1}, \ldots, \frac{1}{2a+1}\}$.
Applying moving averages with $a = 2, 12,$ and 40 to the closing values of our `tui`–dataset implies using following filters:

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

A possible interpretation of these filters are (approximately) weekly ($a = 2$), monthly ($a = 12$) and quaterly ($a = 40$) averages of returns. The fitering is carried out in **R** with the `filter( )` – command.
One way to plot the closing values of the TUI shares and the averages in different colours is to use the following code:

```
library(ts)
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")
```

It creates the following output:

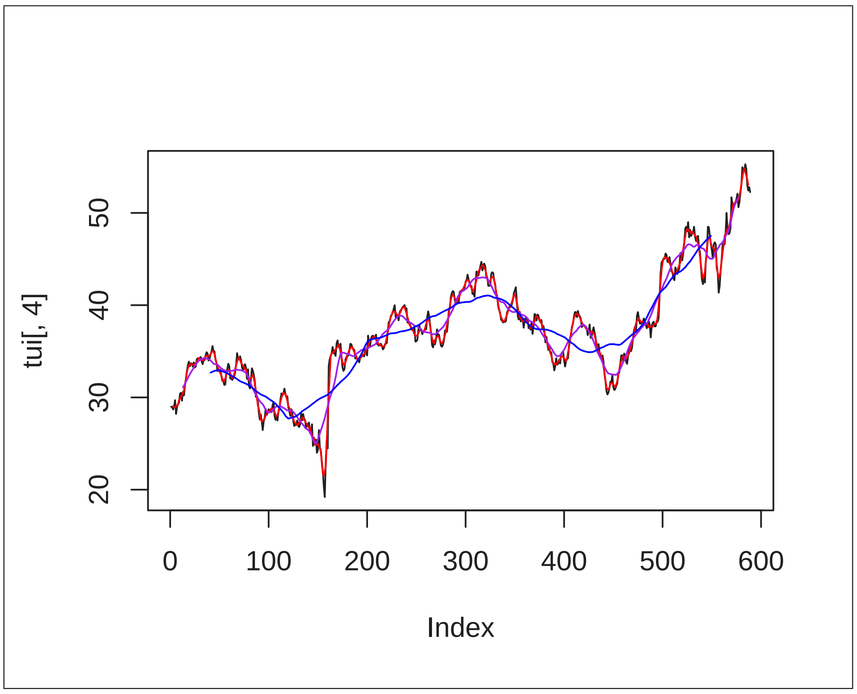

Figure 2.1: Closing values and averages for $a = 2$, 12 and 40

2.2 Decomposition of Time Series

Another possibility for evaluating the trend of a time series is to use nonparametric regression techniques (which in fact can be seen as a special case of linear filters). 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 calculating the seasonal component S_t (and the residuals e_t) from the differences $X_t - T_t$. 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:/beer.csv",header=T,dec=",",sep=";")
beer<-ts(beer[,1],start=1956,freq=12)
plot(stl(log(beer),s.window="periodic"))
```

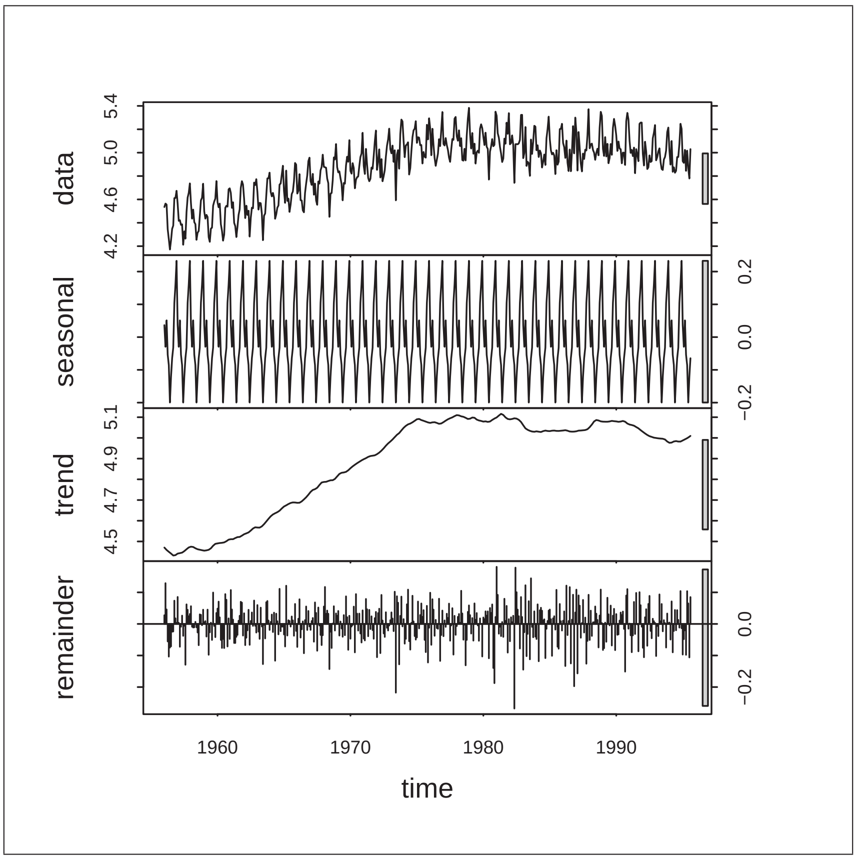

Figure 2.2: Seasonal decomposition using `stl( )`

First, the data is read from `C:/beer.csv` (the datafile is available for download from *http://134.76.173.220/beer.zip* as a zipped csv-file). Then the data is transformed 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.

2.3 Regression analysis

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 wish 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 fit can be carried out in **R** with the following commands:

```
lbeer<-log(beer)
t<-seq(1956,1995.2,length=length(beer))
t2<-t^2
plot(lbeer)
lm(lbeer~t+t2)
lines(lm(lbeer~t+t2)$fit,col=2,lwd=2)
```

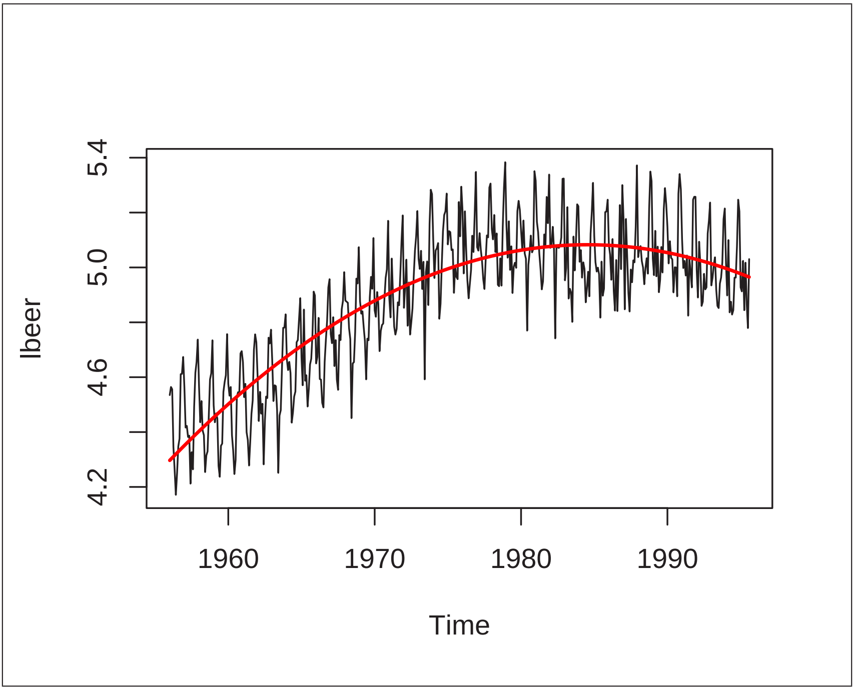

Figure 2.3: Fitting a parabola to `lbeer` using `lm( )`

In the first row of the commands above, logarithms of `beer` are calculated 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

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

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(1956,1995.2,length=length(beer))
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)
```

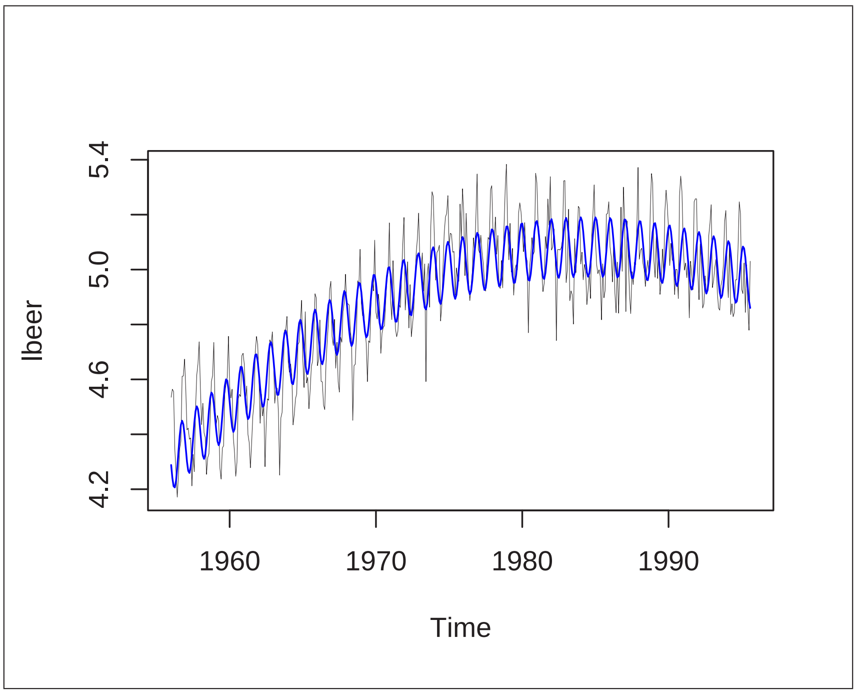

Figure 2.4: Fitting a parabola and the first fourier frequency to `lbeer`

Note that in this case `sin.t` does not include 12 in the denominator, since $\frac{1}{12}$ has already been considered during the transformation of `beer` and the creation 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(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
-0.331911 -0.086555 -0.003136  0.081774  0.345175

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -3.833e+03  1.841e+02 -20.815   <2e-16 ***
t            3.868e+00  1.864e-01  20.751   <2e-16 ***
t2          -9.748e-04  4.718e-05 -20.660   <2e-16 ***
sin.t       -1.078e-01  7.679e-03 -14.036   <2e-16 ***
cos.t       -1.246e-02  7.669e-03  -1.624    0.105
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.1184 on 471 degrees of freedom
Multiple R-Squared: 0.8017,     Adjusted R-squared:   0.8
F-statistic: 476.1 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 would include γ anyway, since Fourier frequencies are always kept in pairs of sine and cosine).

Chapter 3

Exponential Smoothing

3.1 Introductionary Remarks

A natural estimate for predicting the next value of a given time series x_t at the period $t = \tau$ is to take weighted sums of past observations:

$$\hat{x}_{(t=\tau)}(1) = \lambda_0 \cdot x_\tau + \lambda_1 \cdot x_{\tau-1} + \ldots$$

It seems reasonable to weight recent observations more than observations from the past. Hence, one possibility is to use geometric weights

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

such that $\quad \hat{x}_{(t=\tau)}(1) = \alpha \cdot x_\tau + \alpha(1-\alpha) \cdot x_{\tau-1} + \alpha(1-\alpha)^2 \cdot x_{\tau-2} + \ldots$

Exponential smoothing in its basic form (the term "exponential" comes from the fact that the weights decay exponentially) should only be used for 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 containg 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).

3.2 Exponential Smoothing and Prediction of Time Series

The `ts` – library in **R** contains the function `HoltWinters(x,alpha,beta,gamma)`, which lets one perform the Holt–Winters procedure on 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 using `gamma=0`. In case one does not specify smoothing parameters, these are determined "automatically" (i.e. by minimizing the mean squared prediction error from one–step forecasts).

Thus, exponential smoothing of the `beer` – dataset could be performed as follows in **R**:

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

This loads the dataset from the CSV–file and transforms it to a `ts` – object.

```
HoltWinters(beer)
```

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

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

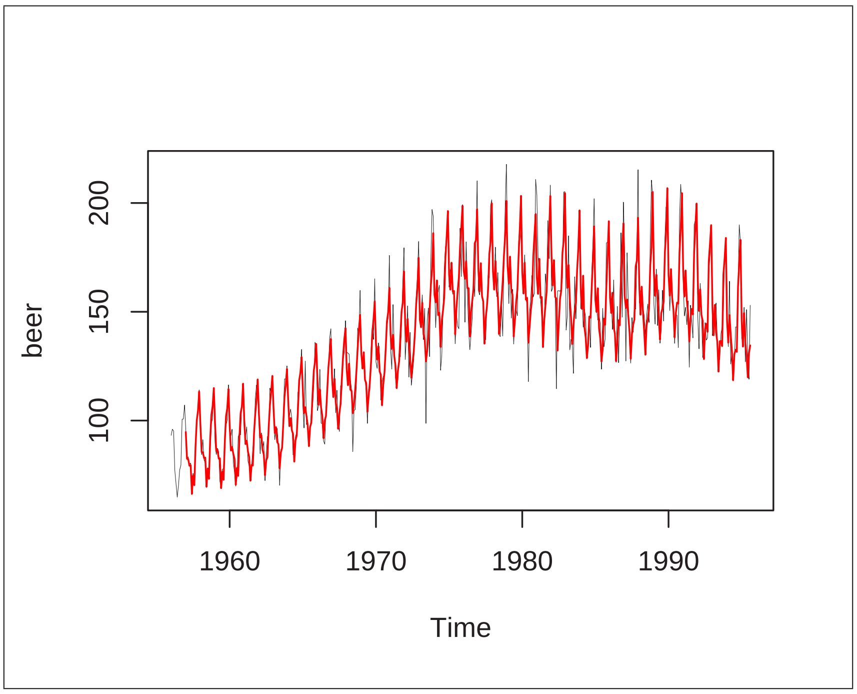

Figure 3.1: Exponential smoothing of the `beer`–dataset

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)
```

gives us 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)
```

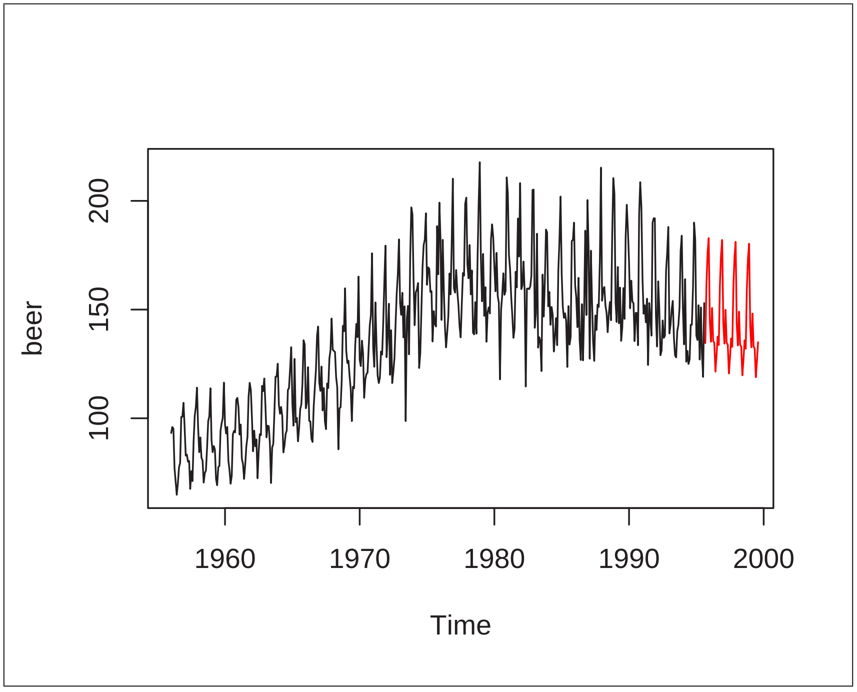

Figure 3.2: Predicting `beer` with exponential smoothing

Chapter 4

ARIMA–Models

4.1 Introductionary Remarks

Forecasting based on ARIMA (autoregressive integrated moving averages) models, commonly know as the Box–Jenkins approach, comprises following stages:

i.) *Model identification*

ii.) *Parameter estimation*

iii.) *Diagnostic checking*

These stages are repeated until a "suitable" model for the given data has been identified (e.g. for prediction). The following three sections show some facilities that **R** offers for assisting the three stages in the Box–Jenkins approach.

4.2 Analysis of Autocorrelations and Partial Autocorrelations

A first step in analyzing 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:

```
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")
```

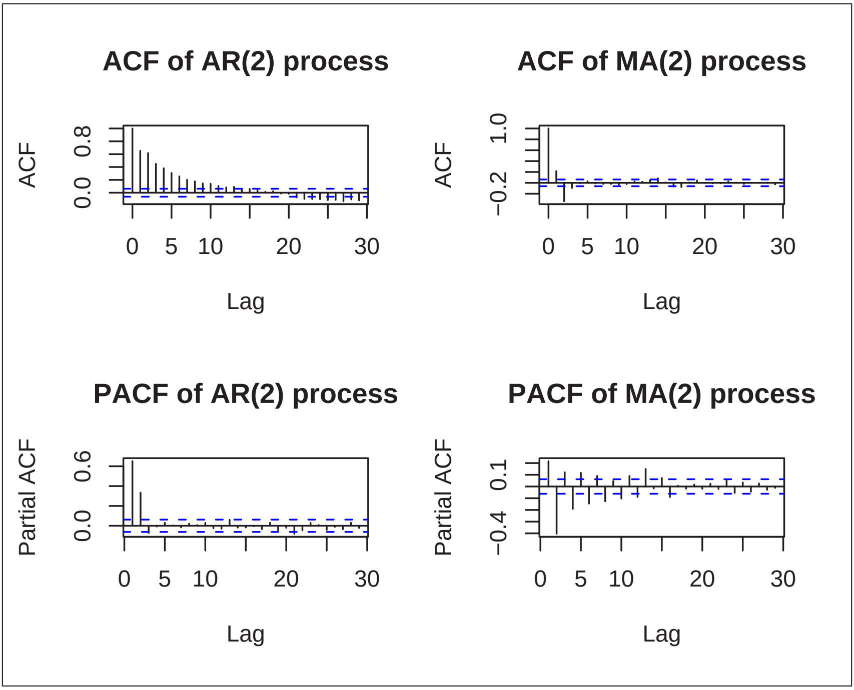

Figure 4.1: ACF and PACF of AR– and MA–models

The function `arima.sim( )` was used to simulate ARIMA(p,d,q)–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 seperate regions within the graphics window.
The last four lines created the ACF and PACF plots of the two simulated processes. Note that by default the plots include confidence intervals (based on uncorrelated series).

4.3 Parameter–Estimation of ARIMA–Models

Once the order of the ARIMA(p,d,q)–model has been specified, the function `arima( )` from the `ts`–library can be used to estimate the parameters:

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

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

4.4 Diagnostic Checking

A first step in diagnostic checking of fitted models is to analyze 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:

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

It produces following output containing a plot of the residuals, the autocorrelation of the residuals and the p-values of the Ljung–Box statistic for the first 10 lags:

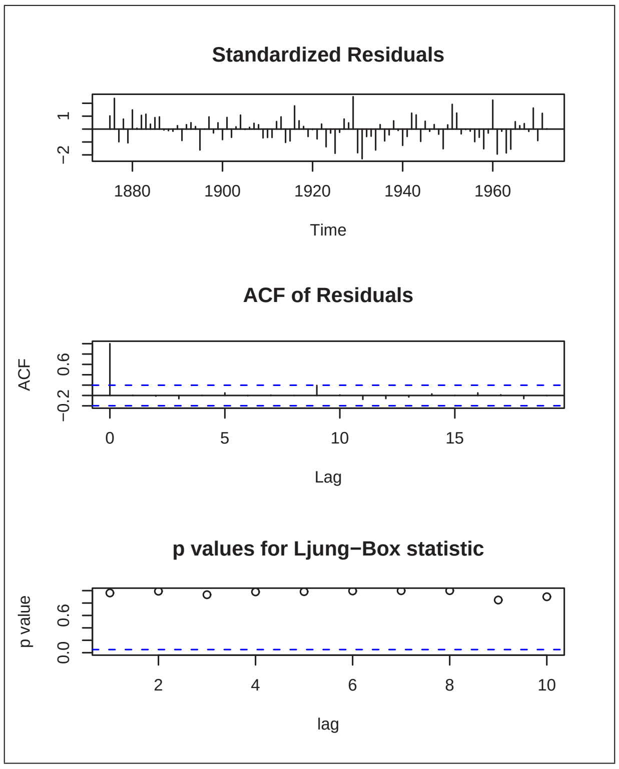

Figure 4.2: Output from `tsdiag`

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(fit$residuals,lag=1)
```

4.5 Prediction of ARIMA–Models

Once a model has been identified and its parameters have been estimated, one purpose is to predict future values of a time series. Lets assume, that we are satisfied with 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:

```
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 a rule of thumb for an approximate confidence interval (95%) of the prediction, *"prediction ± 2·SE"*, one can e.g. plot the Lake Huron data, predicted values and an approximate confidence interval:

```
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. To leave some space for adding the predicted values, the x-axis has been "limited" from 1875 to 1980 with `xlim=c(1875,1980)` ; the use of `ylim` is purely for cosmetic purposes here. The prediction takes place in the second line using `predict( )` on our 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`):

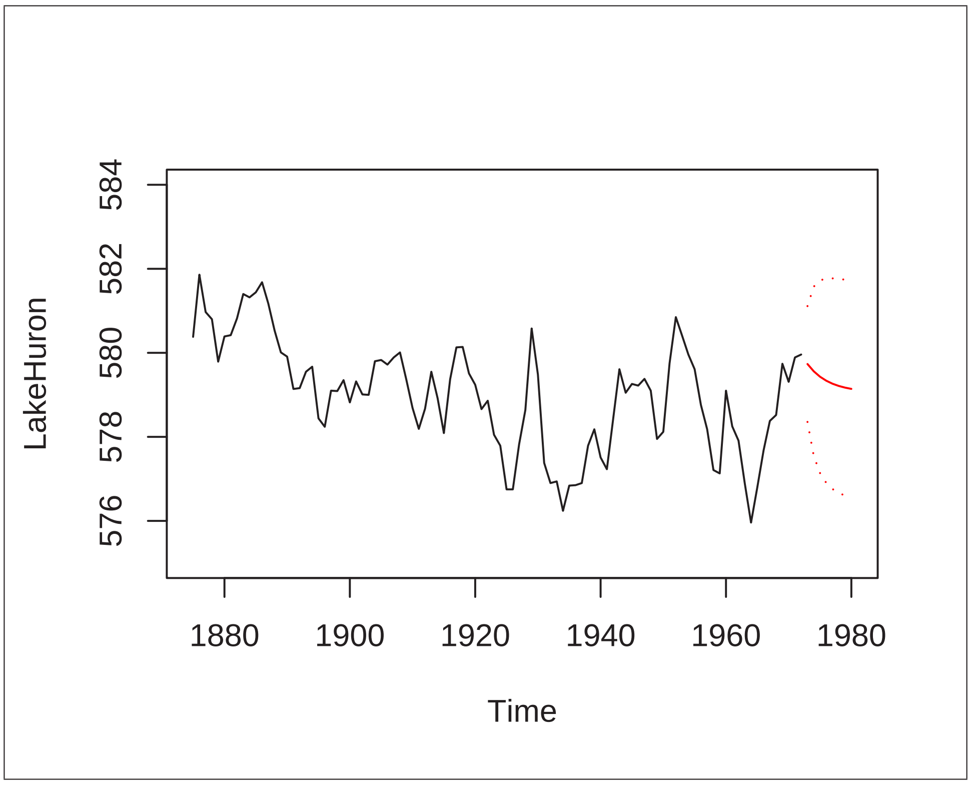

Figure 4.3: Lake Huron levels and predicted values

Appendix A

Function Reference

`abline( )`	Graphics command	*p.*	*6*
`acf( )`	Estimation of the autocorrelation function	*p.*	*17*
`arima( )`	Fitting ARIMA–models	*p.*	*18*
`arima.sim( )`	Simulation of ARIMA–models	*p.*	*17*
`Box.test( )`	Box–Pierce and Ljung–Box test	*p.*	*20*
`c( )`	Vector command	*p.*	*5*
`cos( )`	Cosine	*p.*	*12*
`density( )`	Density estimation	*p.*	*5*
`diff( )`	Takes differences	*p.*	*4*
`dnorm( )`	Normal distribution	*p.*	*6*
`filter( )`	Filtering of time series	*p.*	*9*
`hist( )`	Draws a histogram	*p.*	*5*
`HoltWinters( )`	Holt–Winters procedure	*p.*	*14*
`ks.test( )`	Kolmogorov–Smirnov test	*p.*	*5*
`length( )`	Vector command	*p.*	*12*
`lines( )`	Graphics command	*p.*	*5*
`lm( )`	Linear models	*p.*	*11*
`log( )`	Calculates logs	*p.*	*4*
`lsfit( )`	Least squares estimation	*p.*	*11*
`mean( )`	Calculates means	*p.*	*5*
`pacf( )`	Estimation of the partial autocorrelation function	*p.*	*17*
`plot( )`	Graphics command	*p.*	*3*
`predict( )`	Generic function for prediction	*p.*	*15*
`read.csv( )`	Data import from CSV–files	*p.*	*3*
`rep( )`	Vector command	*p.*	*9*
`sd( )`	Standard deviation	*p.*	*5*
`seq( )`	Vector command	*p.*	*6*
`shapiro.test( )`	Shapiro–Wilk test	*p.*	*7*
`sin( )`	Sine	*p.*	*12*
`stl( )`	Seasonal decomposition of time series	*p.*	*10*
`summary( )`	Generic function for summaries	*p.*	*13*
`ts( )`	Creating time–series objects	*p.*	*10*
`tsdiag( )`	Time series diagnostic	*p.*	*19*
`qqnorm( )`	Quantile–quantile plot	*p.*	*6*