3 Time Series in R

3.1 Time Series Classes

In **R**, there are *objects*, which are organized in a large number of *classes*. These classes e.g. include *vectors*, *data frames*, *model output*, *functions*, and many more. Not surprisingly, there are also several classes for time series. We start by presenting `ts`, the basic class for regularly spaced time series. This class is comparably simple, as it can only represent time series with fixed interval records, and only uses numeric time stamps, i.e. (sophistically) enumerates the index set. However, it will be sufficient for most, if not all, of what we do in this course. Then, we also provide an outlook to more complicated concepts.

3.1.1 The ts Class

For defining a time series of class `ts`, we of course need to provide the *data*, but also the *starting time* as argument `start`, and the *frequency* of measurements as argument `frequency`. If no starting time is supplied, **R** uses its default value of 1, i.e. enumerates the times by the index set $1, ..., n$, where n is the length of the series. The frequency is the number of observations per unit of time, e.g. 1 for yearly, 4 for quarterly, or 12 for monthly recordings. Instead of the start, we could also provide the end of the series, and instead of the frequency, we could supply argument `deltat`, the fraction of the sampling period between successive observations. The following example will illustrate the concept.

Example: We here consider a simple and short series that holds the number of days per year with traffic holdups in front of the Gotthard road tunnel north entrance in Switzerland. The data are available from the Federal Roads Office.

2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
88	76	112	109	91	98	139	150	168	149

The start of this series is in 2004. The time unit is years, and since we have just one record per year, the frequency of this series is 1. This tells us that while there may be a trend, there cannot be a seasonal effect, as the latter can only be present in periodic series, i.e. series with frequency > 1. We now define a `ts` object in in **R**.

```
> rawdat <- c(88, 76, 112, 109, 91, 98, 139, 150, 168, 149)
> ts.dat <- ts(rawdat, start=2004, freq=1)
> ts.dat
Time Series: Start = 2004, End = 2013
Frequency = 1
[1]  88  76 112 109  91  98 139 150 168 149
```

There are a number of simple but useful functions that extract basic information from objects of class `ts`, see the following examples:

```
> start(ts.dat)
[1] 2004    1

> end(ts.dat)
[1] 2013    1

> frequency(ts.dat)
[1] 1

> deltat(ts.dat)
[1] 1
```

Another possibility is to obtain the measurement times from a time series object. As class `ts` only enumerates the times, they are given as fractions. This can still be very useful for specialized plots, etc.

```
> time(ts.dat)
Time Series:
Start = 2004
End = 2013
Frequency = 1
[1] 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013
```

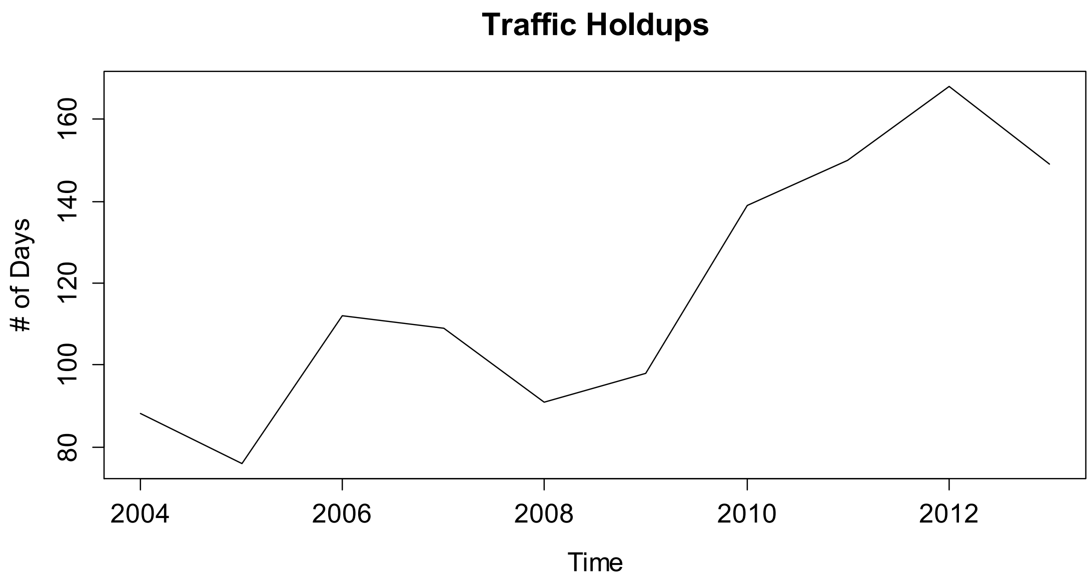
The next basic, but for practical purposes very useful function is `window()`. It is aimed at selecting a subset from a time series. Of course, also regular **R**-subsetting such as `ts.dat[2:5]` does work with the time series class. However, this results in a vector rather than a time series object, and is thus mostly of less use than the `window()` command.

```
> window(ts.dat, start=2006, end=2008)
Time Series:
Start = 2006
End = 2008
Frequency = 1
[1] 112 109  91
```

While we here presented the most important basic methods/functions for class `ts`, there is a wealth of further ones. This includes the `plot()` function, and many more, e.g. for estimating trends, seasonal effects and dependency structure, for fitting time series models and generating forecasts. We will present them in the forthcoming chapters of this scriptum.

To conclude the previous example, we will not do without showing the time series plot of the Gotthard road tunnel traffic holdup days, see next page. Because there are a limited number of observations, it is difficult to give statements regarding a possible trend and/or stochastic dependency.

```
> plot(ts.dat, ylab="# of Days", main="Traffic Holdups")
```



3.1.2 Finding the Frequency

Often, finding the frequency for defining the time series is straightforward. As mentioned above, it is the number of observations per unit of time, e.g. 1 for yearly, 4 for quarterly, or 12 for monthly recordings. However, some real-world situation are quite a bit more complex to handle. Principally, it is up to the user to choose the correct frequency from background and field knowledge about the measurements. Additionally, R function `findfrequency()` may assist. It provides the correct results for the traffic holdups and the air passenger data:

```
> findfrequency(ts.dat)
[1] 1
> findfrequency(AirPassengers)
[1] 12
```

On the other hand, it can also provide misleading results. If we apply the function to the lynx data, we obtain:

```
> findfrequency(lynx)
[1] 10
```

The lynx data are clearly cyclic with a period of about 10 years. We interpret these cycles as stochastic though and the frequency should be set to a value of 1. In other cases, there may be ambiguity in the definition of the frequency. If we for example consider the a time series of the minutely averaged electricity demand in a city, the frequency may be:

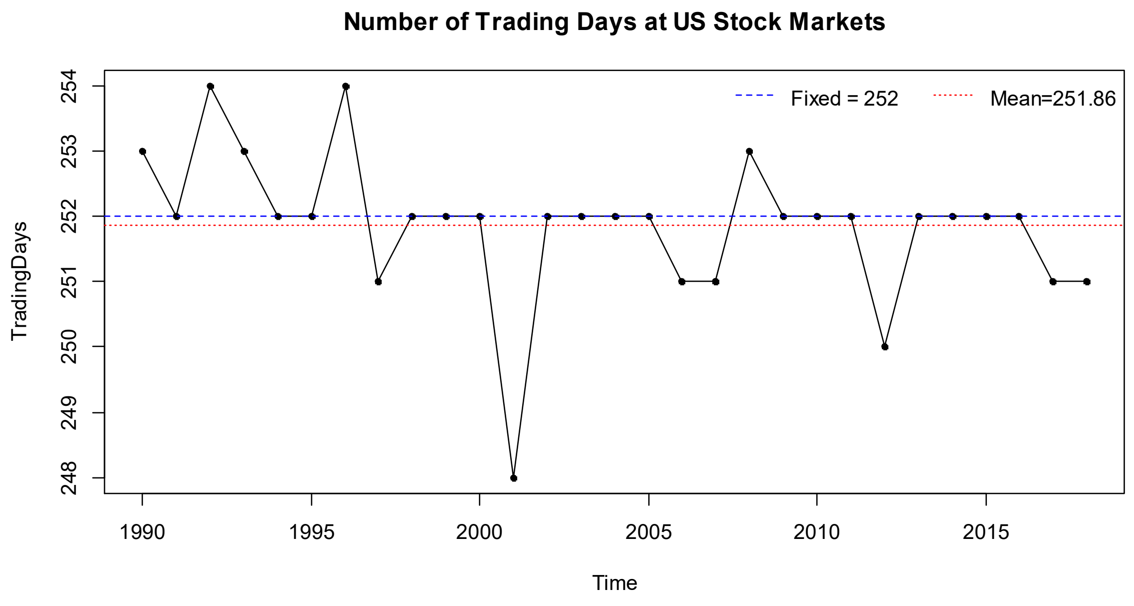
- Hourly (i.e. $f = 60$)
- Daily (i.e. $f = 24 \cdot 60 = 1'440$)
- Weekly (i.e. $f = 24 \cdot 60 \cdot 7 = 10'080$)
- Yearly (i.e. $f = 24 \cdot 60 \cdot 365 = 525'600$)

When working with the `ts()` class, we need to decide for one single frequency. That may be far from easy, because the power demand may have a hourly, daily, weekly and yearly pattern. The simple rule of the thumb is to pick the frequency which is the most natural, the strongest or the most central for the analysis which is carried out. Sometimes (*especially for providing accurate results in forecasting*), all cyclic components need to be kept under the radar. For achieving this, clever decomposition approaches may help. Moreover, there is the advanced `msts()` class in R that allows time series to have multiple "frequencies". As soon as one extensively deals with weekly or daily data, further problems will appear. Namely, the number of observations per time unit may not be constant or not an integer.

- *Weekly data*: even in the simple case where all (observation) years have exactly 365 days, we obtain a non-integer frequency of $f = 365/7 = 52.14$.

- *Daily data:* the problem here arises from the leap years that have 366 days. As they (roughly) happen every 4th year, the quick fix is to set $f = 365.25$. This is still somewhat imprecise, because the leap year rules are more complicated and sometimes leap seconds are used, altering the astronomically correct frequency slightly. In most practical cases (if the time series does not comprise of hundreds or thousands of observations years) it won't make much practical difference, though.

- *Trading or working day data:* the number of working days per year fluctuates even more, so that the definition of the frequency becomes tricky. One either works with a "representative" integer value or the mean resp. median number of working days per year. Most R functions also accept non-integer frequency values, making this strategy viable. Below we have the number of trading days at US Stock Markets. Commong knowledge says that "it's usually 252 trading days a year", suggesting this value for the frequency. Alternatively, the mean of 251.86 could be used.



3.1.3 Other Classes

Besides the basic `ts` class, there are several other classes which offer a variety of additional options. Most are designed for specific and advanced tasks, so that they will rarely to never be required during our course. Most prominently, this includes the `zoo` package, which provides infrastructure for both regularly and irregularly spaced time series using arbitrary classes for the time stamps. It is designed to be as consistent as possible with the `ts` class. Coercion from and to `zoo` is also readily available.

Some further packages which contain classes and methods for time series include `xts`, `its`, `tseries`, `fts`, `timeSeries` and `tis`. Additional information on their content and philosophy can be found on CRAN.

3.2 Dates and Times in R

While for the `ts` class, the handling of times has been solved very simply and easily by enumerating, doing time series analysis in R may sometimes also require to explicitly working with date and time. There are several options for dealing with date and date/time data. The built-in `as.Date()` function handles dates that come without times. The contributed package `chron` handles dates and times, but does not control for different time zones, whereas the sophisticated but complex `POSIXct` and `POSIXlt` classes allow for dates and times with time zone control.

As a general rule for date/time data in **R**, we suggest to use the simplest technique possible. Thus, for date only data, `as.Date()` will mostly be the optimal choice. If handling dates and times, but without time-zone information, is required, the `chron` package is the choice. The `POSIX` classes are especially useful in the relatively rare cases when time-zone manipulation is important.

Apart for the `POSIXlt` class, dates/times are internally stored as the number of days or seconds from some reference date. These dates/times thus generally have a numeric mode. The `POSIXlt` class, on the other hand, stores date/time values as a list of components (`hour`, `min`, `sec`, `mon`, etc.), making it easy to extract these parts. Also the current date is accessible by typing `Sys.Date()` in the console, and returns an object of class `Date`.

3.2.1 The Date Class

As mentioned above, the easiest solution for specifying days in R is with the `as.Date()` function. Using the format argument, arbitrary date formats can be read. The default, however, is four-digit year, followed by month and then day, separated by dashes or slashes:

```
> as.Date("2012-02-14")
[1] "2012-02-14"
```

```
> as.Date("2012/02/07")
[1] "2012-02-07"
```

If the dates come in non-standard appearance, we require defining their format using some codes. While the most important ones are shown below, we reference to the **R** help file of function `strptime()` for the full list.

Code	**Value**
%d	Day of the month (decimal number)
%m	Month (decimal number)
%b	Month (character, abbreviated)
%B	Month (character, full name)
%y	Year (decimal, two digit)
%Y	Year (decimal, four digit)

The following examples illustrate the use of the `format` argument:

```
> as.Date("27.01.12", format="%d.%m.%y")
[1] "2012-01-27"
> as.Date("14. Februar, 2012", format="%d. %B, %Y")
[1] "2012-02-14"
```

Internally, Date objects are stored as the number of days passed since the 1st of January in 1970. Earlier dates receive negative numbers. By using the `as.numeric()` function, we can easily find out how many days are past since the reference date. Also back-conversion from a number of past days to a date is straightforward:

```
> mydat <- as.Date("2012-02-14")
> ndays <- as.numeric(mydat)
> ndays
[1] 15384
> tdays <- 10000
> class(tdays) <- "Date"
> tdays
[1] "1997-05-19"
```

A very useful feature is the possibility of extracting weekdays, months and quarters from `Date` objects, see the examples below. This information can be converted to factors. In this form, they serve for purposes such as visualization, decomposition, or time series regression.

```
> weekdays(mydat)
[1] "Dienstag"
> months(mydat)
[1] "Februar"
> quarters(mydat)
[1] "Q1"
```

Furthermore, some very useful summary statistics can be generated from `Date` objects: `median`, `mean`, `min`, `max`, `range`, ... are all available. We can even subtract two dates, which results in a `difftime` object, i.e. the time difference in days.

```
> dat <- as.Date(c("2000-01-01","2004-04-04","2007-08-09"))
> dat
[1] "2000-01-01" "2004-04-04" "2007-08-09"

> min(dat)
[1] "2000-01-01"
> max(dat)
[1] "2007-08-09"
> mean(dat)
[1] "2003-12-15"
> median(dat)
[1] "2004-04-04"

> dat[3]-dat[1]
Time difference of 2777 days
```

Another option is generating time sequences. For example, to generate a vector of 12 dates, starting on August 3, 1985, with an interval of one single day between them, we simply type:

```
> seq(as.Date("1985-08-03"), by="days", length=12)
 [1] "1985-08-03" "1985-08-04" "1985-08-05" "1985-08-06"
 [5] "1985-08-07" "1985-08-08" "1985-08-09" "1985-08-10"
 [9] "1985-08-11" "1985-08-12" "1985-08-13" "1985-08-14"
```

The `by` argument proves to be very useful. We can supply various units of time, and even place an integer in front of it. This allows creating a sequence of dates separated by two weeks:

```
> seq(as.Date("1992-04-17"), by="2 weeks", length=12)
 [1] "1992-04-17" "1992-05-01" "1992-05-15" "1992-05-29"
 [5] "1992-06-12" "1992-06-26" "1992-07-10" "1992-07-24"
 [9] "1992-08-07" "1992-08-21" "1992-09-04" "1992-09-18"
```

3.2.2 The chron Package

The `chron()` function converts dates and times to `chron` objects. The dates and times are provided separately to the `chron()` function, which may well require some inital pre-processing. For such parsing, **R**-functions such as `substr()` and `strsplit()` can be of great use. In the `chron package`, there is no support for time zones and daylight savings time, and `chron` objects are internally stored as fractional days since the reference date of January 1st, 1970. By using the function `as.numeric()`, these internal values can be accessed. The following example illustrates the use of `chron`:

```
> library(chron)
> dat <- c("2007-06-09 16:43:20", "2007-08-29 07:22:40",
           "2007-10-21 16:48:40", "2007-12-17 11:18:50")
> dts <- substr(dat,  1, 10)
> tme <- substr(dat, 12, 19)
> fmt <- c("y-m-d","h:m:s")
> cdt <- chron(dates=dts, time=tme, format=fmt)
> cdt
[1] (07-06-09 16:43:20) (07-08-29 07:22:40)
[3] (07-10-21 16:48:40) (07-12-17 11:18:50)
```

As before, we can again use the entire palette of summary statistic functions. Of some special interest are time differences, which can now be obtained as either fraction of days, or in weeks, hours, minutes, seconds, etc.:

```
> cdt[2]-cdt[1]
Time in days:
[1] 80.61065
> difftime(cdt[2], cdt[1], units="secs")
Time difference of 6964760 secs
```

3.2.3 POSIX Classes

The two classes `POSIXct` and `POSIXlt` implement date/time information, and in contrast to the `chron` package, also support time zones and daylight savings time. We recommend utilizing this functionality only when urgently needed, because the handling requires quite some care, and may on top of that be system dependent. Further details on the use of the POSIX classes can be found on CRAN.

As explained above, the `POSIXct` class also stores dates/times with respect to the internal reference, whereas the `POSIXlt` class stores them as a list of components (`hour`, `min`, `sec`, `mon`, etc.), making it easy to extract these parts.

3.3 Data Import

We can safely assume that most time series data are already present in electronic form; however, not necessarily in **R**. Thus, some knowledge on how to import data into **R** is required. It is be beyond the scope of this scriptum to present the uncounted options which exist for this task. Hence, we will restrict ourselves to providing a short overview and some useful hints.

The most common form for sharing time series data are certainly spreadsheets, or in particular, Microsoft Excel files. While `library(ROBDC)` offers functionality to directly import data from Excel files, we discourage its use. First of all, this only works on Windows systems. More importantly, it is usually simpler, quicker and more flexible to export comma- or tab-separated text files from Excel, and import them via the ubiquitous `read.table()` function, respectively the tailored version `read.csv()` (for comma separation) and `read.delim()` (for tab separation).

With packages `ROBDC` and `RMySQL`, **`R`** can also communicate with SQL databases, which is the method of choice for large scale problems. Furthermore, after loading `library(foreign)`, it is also possible to read files from Stata, SPSS, Octave and SAS.

4 Descriptive Analysis

As always when working with data, i.e. “a pile of numbers”, it is important to gain an overview. In time series analysis, this encompasses several aspects:

- understanding the context of the problem and the data source
- making suitable plots, looking for general structure and outliers
- thinking about data transformations, e.g. to reduce skewness
- judging stationarity and potentially achieve it by decomposition
- for stationary series, the analysis of the autocorrelation function

We start by discussing time series plots, then discuss transformations, focus on the decomposition of time series into trend, seasonal effect and stationary random part and conclude by discussing methods for visualizing the dependency structure.

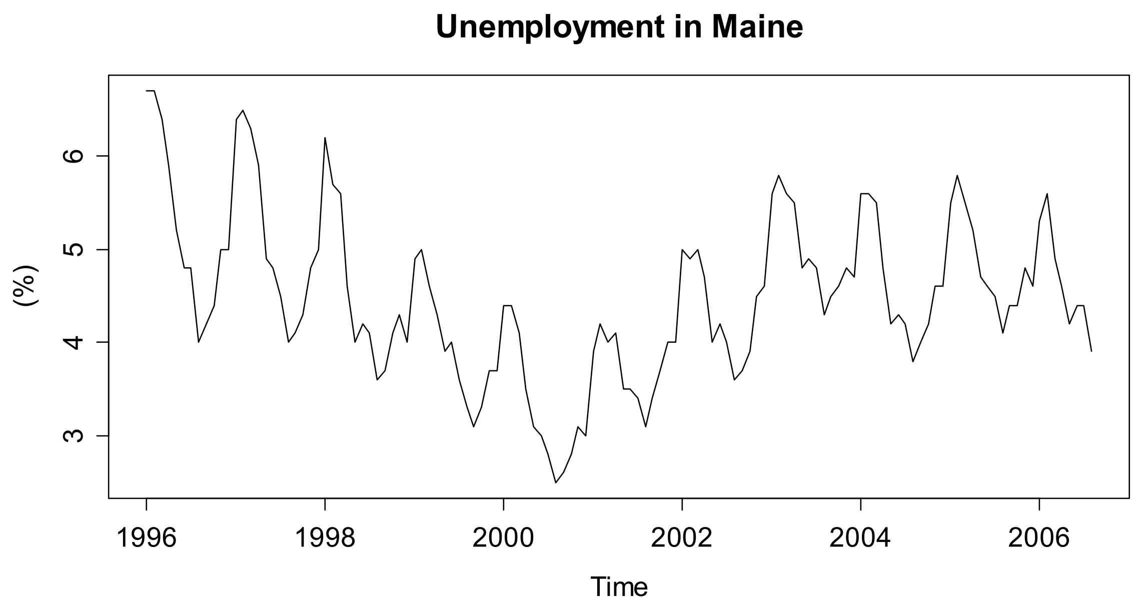
4.1 Visualization

4.1.1 Time Series Plot

The most important means of visualization is the time series plot, where the data are plotted versus time/index. There are several examples in section 1.2, where we also got acquainted with **R**’s generic `plot()` function. As a general rule, the data points are joined by lines in time series plots. This is despite the data are not continuous, as the plots are much easier to read in this form. The only exception where gaps are left is if there are missing values. Moreover, the reader expects that the axes are well-chosen, labeled and the measurement units are given.

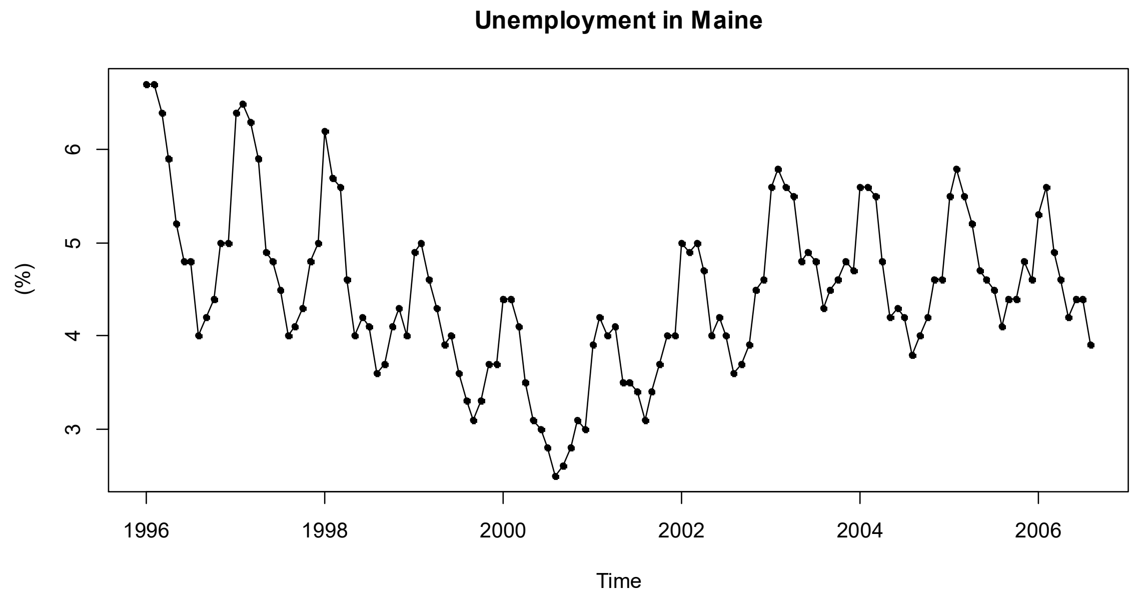
Another issue is the correct aspect ratio for time series plots: if the time axis gets too much compressed, it can become difficult to recognize the behavior of a series. Thus, we recommend choosing the aspect ratio appropriately. However, there are no hard and simple rules on how to do this. As a rule of the thumb, use the “banking the angle to 45 degrees” paradigm: increase and decrease in periodic series should not be displayed at angles much higher or lower than 45 degrees. For very long series, this can become difficult on either A4 paper or a computer screen. In this case, we recommend splitting up the series and display it in different frames. For illustration, we here show an example, the monthly unemployment rate in the US state of Maine, from January 1996 until August 2006. The data originate from the book "Introductory Time Series with R" by Cowpertwait & Metcalfe (https://www.springer.com/gp/book/9780387886978). The website has a zipped folder where you can find `Maine.dat`. Or alternatively, uses the lecturers preprocessed file `unemployment.rda`.

```
> load("unemployment.rda")
> plot(unemp, ylab="(%)", main="Unemployment in Maine")
```



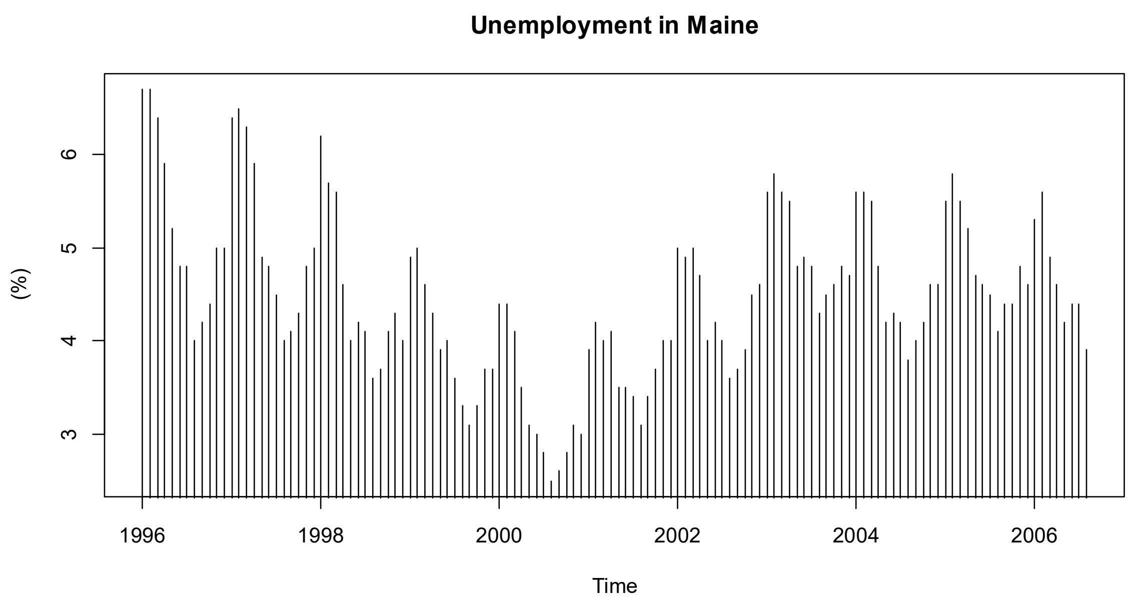
Not surprisingly for monthly economic data, the series shows both a non-linear trend and a seasonal pattern that increases with the level of the series. Hence, using a log-tranformation as explained in section 4.2 may be adviseable. Since unemployment rates are one of the main economic indicators used by politicians/decision makers, this series poses a worthwhile forecasting problem.

```
> plot(unemp, type="o", pch=20, ylab="(%)", main="…")
```



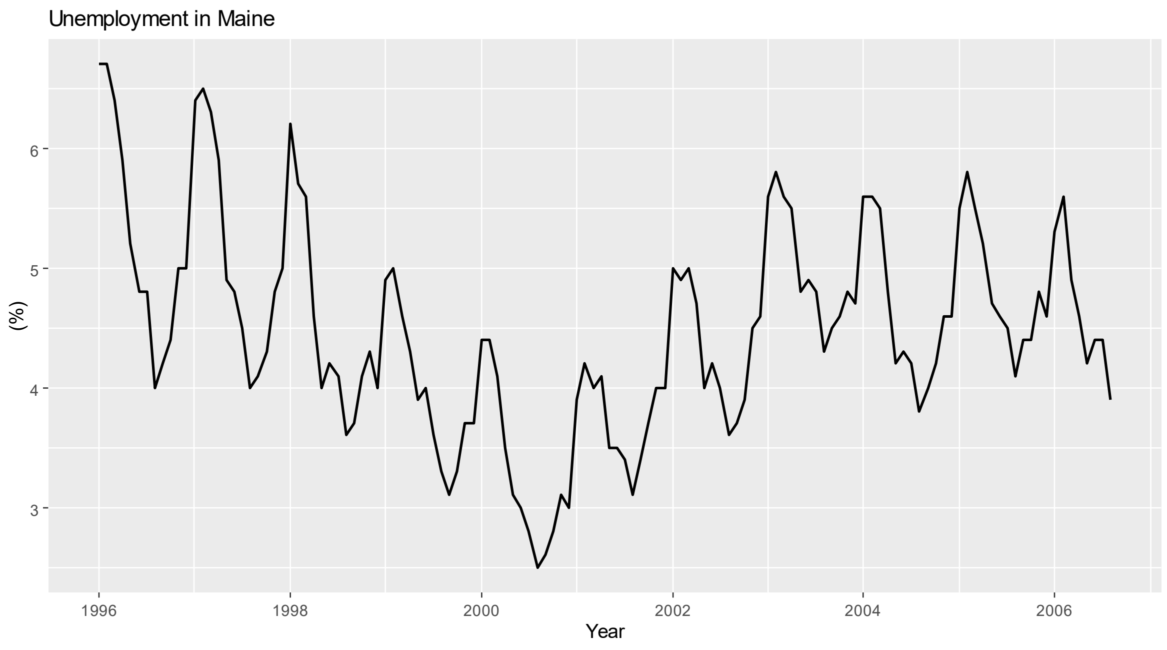
There are various ways by which time series plots can be enhanced. In some cases when only relatively few data points are present and there is no distinct seasonal pattern, "adding the points" with argument `type="o"` may be worthwhile. In other applications, it has become the quasi-norm to plot vertical lines by `type="h"` rather than the time series plot shown above.

```
> plot(unemp, type="h", ylab="(%)", main="…")
```



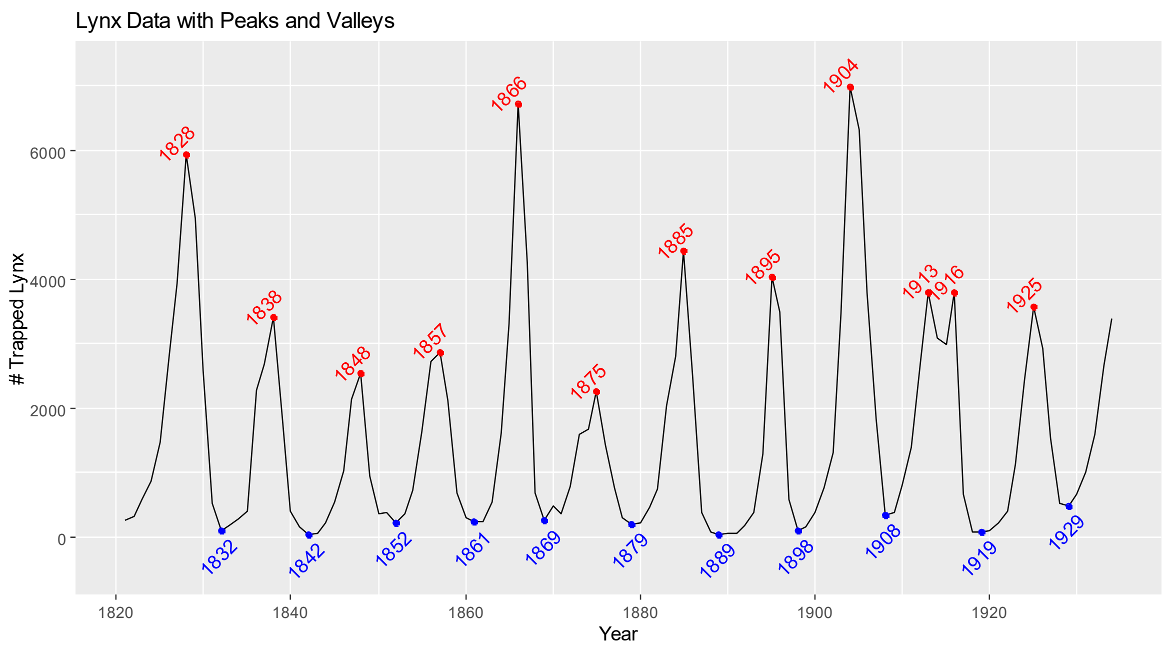
In recent years, the `ggplot2`-package in R with it's elegant graphics has become very popular. Generating a simple time series plot requires a bit more effort than with R standard graphics, but the main advantage lies in the numerous enhancements for complex data analysis situations that are relatively straightforward. It is however beyond the scope of this course to give extensive details about `ggplot2` and unless needed, we will work with R standard graphics throughout this script.

```
> ggplot(unemp, as.numeric=FALSE) + geom_line(size=1) +
+   ggtitle("Unemp…") + xlab("Year") + ylab("(%)")
```



We finish this chapter with an automatically annoted time series plot of the lynx data that was generated with an add-on package to `ggplot2`. Producing fancy graphics has become its own discipline in data science and is certainly worthwhile.

```
> ggplot(lynx, as.numeric = FALSE) + geom_line() +
    ggtitle("Lynx Data with Peaks and Valleys") +
    stat_peaks(colour = "red") +
    stat_peaks(geom = "text", colour = "red",
               vjust = -0.5, x.label.fmt = "%Y") +
    stat_valleys(colour = "blue") +
    stat_valleys(geom = "text", colour = "blue", angle = 45,
                 vjust = 1.5, hjust = 1, x.label.fmt = "%Y")+
    ylim(-500, 7300) + xlab("Year") + ylab("# Trapped Lynx")
```



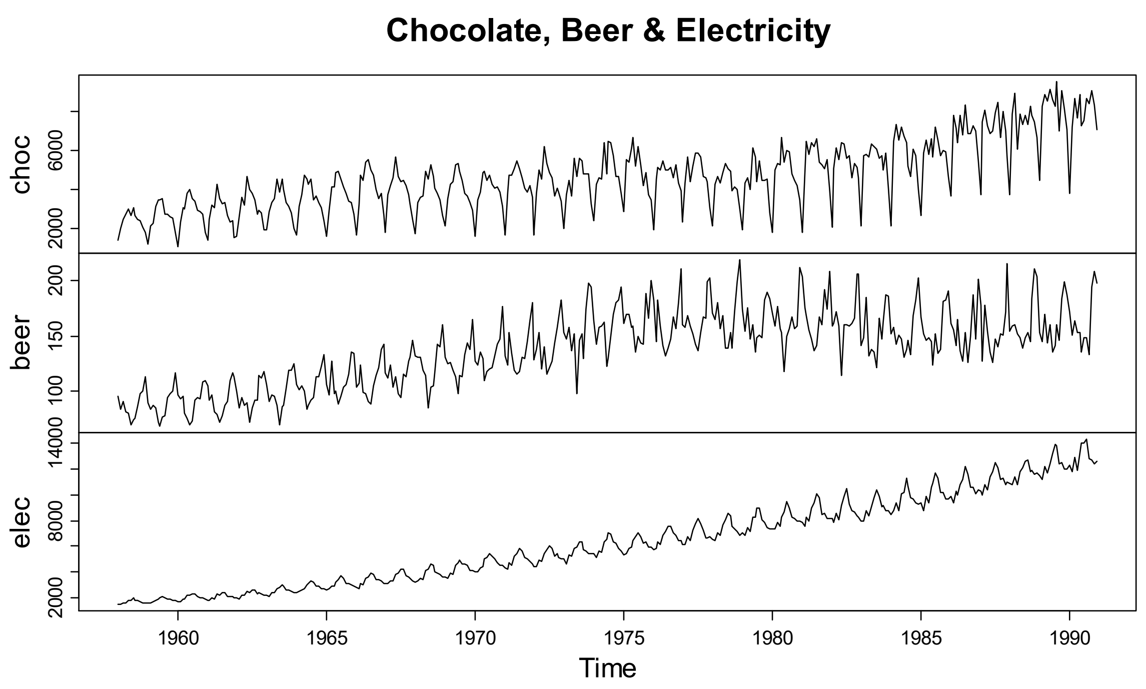
4.1.2 Multiple Time Series Plots

In applied problems, one is sometimes provided with multiple time series. Here, we illustrate some basics on import, definition and plotting. Our example exhibits the monthly supply of electricity (millions of kWh), beer (millions of liters) and chocolate-based production (tonnes) in Australia over the period from January 1958 to December 1990. These data were published by the Bureau of Australian Statistics and are presented in the book of Cowpertwait & Metcalfe.

```
> dat <- read.table("cbe.dat",sep="", header=T)
> cbe <- ts(dat, start=1958, freq=12)
```

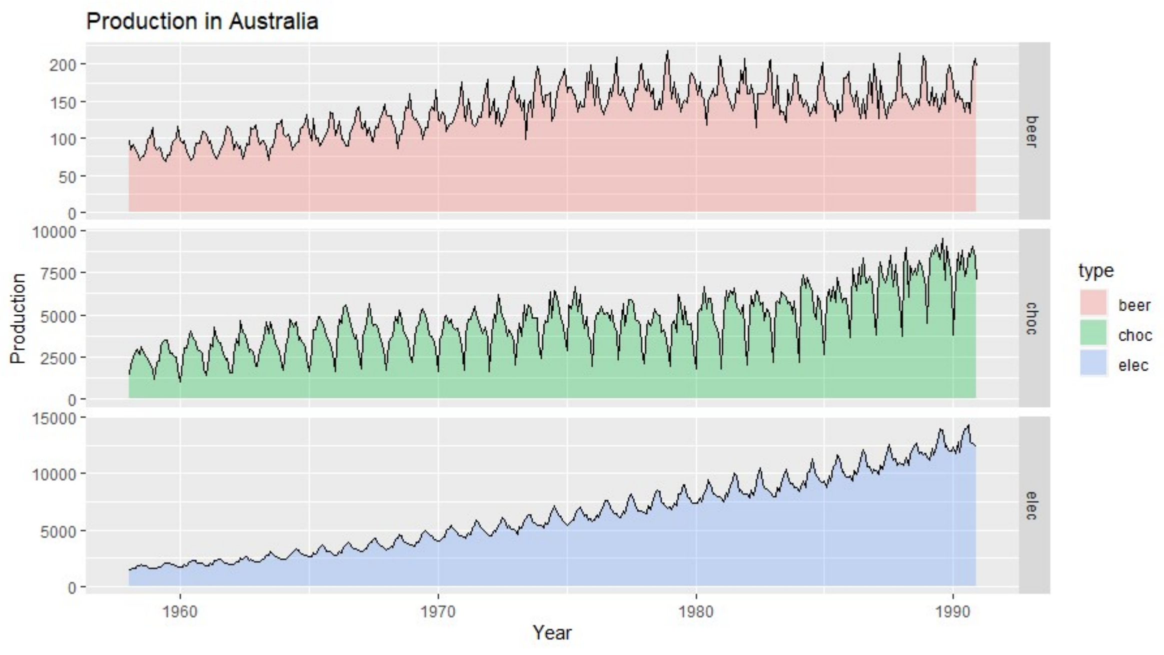
This creates a multiple time series object that can very easily be displayed using the generic plot command again, although the presentation turns out to be a bit dull, see next page.

```
> plot(tsd, main="Chocolate, Beer & Electricity")
```



A much nicer plot can be produced using the `ggplot2`-package using different facets for the three series. However, it does not directly work with the multiple time series object as the input, but requires creating a data frame in long format.

```
> cbedf <- data.frame(t=rep(as.numeric(time(cbe)), times=3),
         values=c(cbe[,1], cbe[,2], cbe[,3]),
         type=rep(c("choc", "beer", "elec"), each=nrow(cbe)))
> ggplot(cbedf, aes(time, values, fill=type)) +
    geom_area(alpha=0.3, size=1) + geom_line() +
    facet_grid(type~., scales="free") +
    ggtitle("Production in Australia") +
    xlab("Year") + ylab("Production")
```

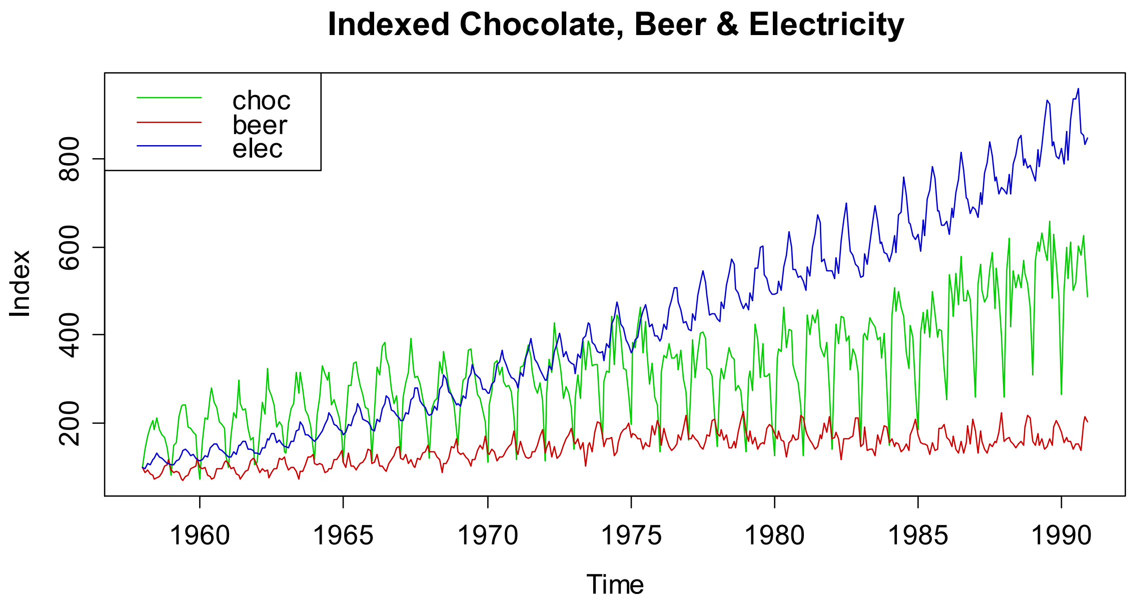


All three series show a distinct seasonal pattern, along with a trend. It is also instructive to know that the Australian population increased by a factor of 1.8 during the period where these three series were observed. As a general rule, using different frames for multiple series is the most recommended means of visualization. However, sometimes it can be more instructive to have them in the same frame. Of course, this requires that the series are either on the same scale, or have been indexed, resp. standardized to be so. Then, we can simply use `plot(ind.tsd, plot.type="single")`. When working with one single panel, we recommend to use different colors for the series, which is easily possible using a `col=c("green3", "red3", "blue3")` argument.

```
## Indexing the series
tsd      <- cbe
tsd[,1] <- tsd[,1]/tsd[1,1]*100
tsd[,2] <- tsd[,2]/tsd[1,2]*100
tsd[,3] <- tsd[,3]/tsd[1,3]*100

## Plotting in one single frame
clr <- c("green3", "red3", "blue3")
plot(tsd, plot.type="single", ylab="Index", col=clr)
title("Indexed Chocolate, Beer & Electricity")

## Legend
ltxt <- names(dat)
legend("topleft", lty=1, col=clr, legend=ltxt)
```



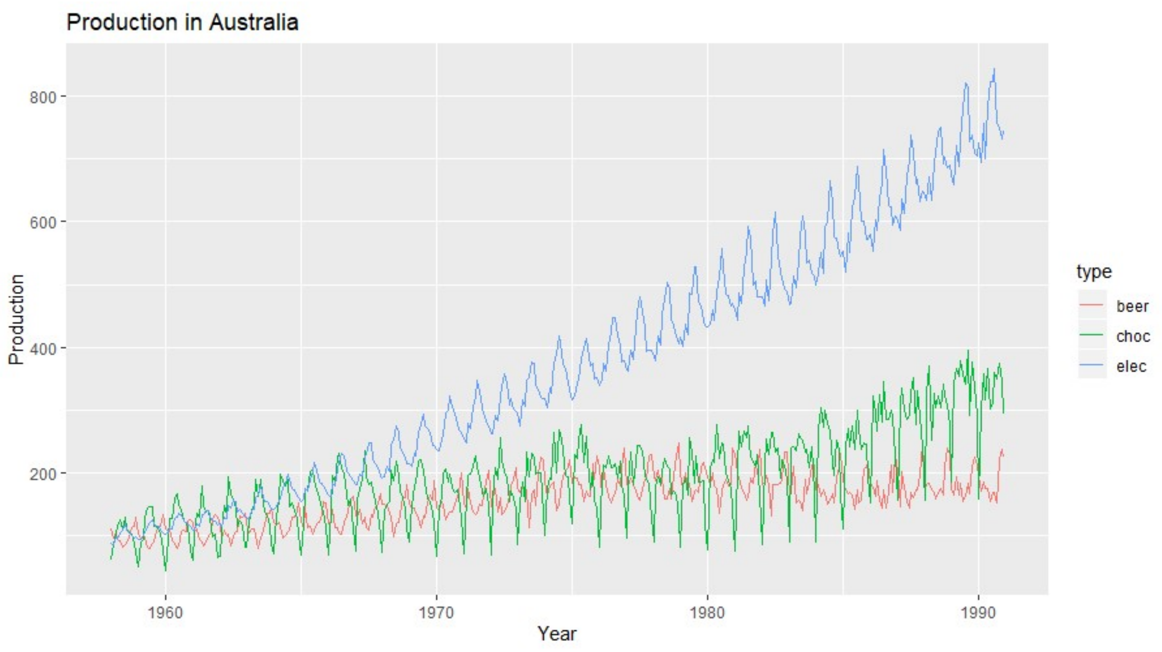
In the indexed single frame plot above, we can very well judge the relative development of the series over time. Due to different scaling, this was nearly impossible with the multiple frames on the previous page. We observe that electricity production increased around 8x during 1958 and 1990, whereas for chocolate the multiplier is around 4x, and for beer less than 2x. Also, the seasonal variation is most pronounced for chocolate, followed by electricity and then beer.

A special remark needs to be made about the indexing. In the bit of code above, the series were standardized using their first observed value. For seasonal time series, this may be a suboptimal strategy as one of the series may have the highpoint at the start observation, whereas for another series with an opposite pattern it may be the lowpoint. In such cases, it is usually beneficial to take the entire first period as the reference, i.e.:

```
> ## Indexing the series vs. the first period
> tsd     <- cbe
> tsd[,1] <- tsd[,1]/mean(tsd[1:12,1])*100
> tsd[,2] <- tsd[,2]/mean(tsd[1:12,2])*100
> tsd[,3] <- tsd[,3]/mean(tsd[1:12,3])*100
```

For complementing this chapter about visualization of time series, we present another output that was produced with `ggplot2`. As is typical for this package, adding different colors, legends et cetera, i.e. enhancing the basic plot is straightforward (if you know how to do so) and requires less code than the standard plots in R.

```
> ## Graphical display with ggplot
> ggplot(cbedf, aes(time, values, color=type)) +
    geom_line() +
    ggtitle("Production in Australia") +
    xlab("Year") +
    ylab("Production")
```



Qualitatively, there is also a marked difference between the first version of the plot using only the first value as the reference vs. this later one that standardizes with the first year, notably for the evolution of beer and chocolate consumption. We conclude the chapter by emphasizing that graphical displays of time series should be well chosen and reflected.

4.2 Transformations

Time series data do not necessarily need to be analyzed in the form they were provided to us. In many cases, it is much better, more efficient and instructive to transform the data. We will here highlight several cases and discuss their impact on the results.

4.2.1 Linear Transformations

A linear transformation is of the form $Y_t = a + bX_t$. Examples include simple changes in units, e.g. from meters to kilometers, kilograms to tons, et cetera. Also slightly more complicated conversions as for example brining Fahrenheit temperatures to the Celsius scale fall under this definition. It is obvious that such linear transformations will not change the appearance of the series. Hence, all derived results (i.e. autocorrelations, models, forecasts) will be equivalent. As a consequence, we are free to perform linear transformations whenever it seems convenient.

4.2.2 Calendar Adjustments

Often in time series analysis we consider monthly data and often these are delivered as monthly totals. However, this adds unnecessary noise to the series, simply because of the different number of days per month. Often, the seasonal effect becomes much cleaner and easier to understand if we switch to the daily average per month rather than considering the monthly total. We consider an example where the quantity of interest is the monthly milk production of a cow. The first option is

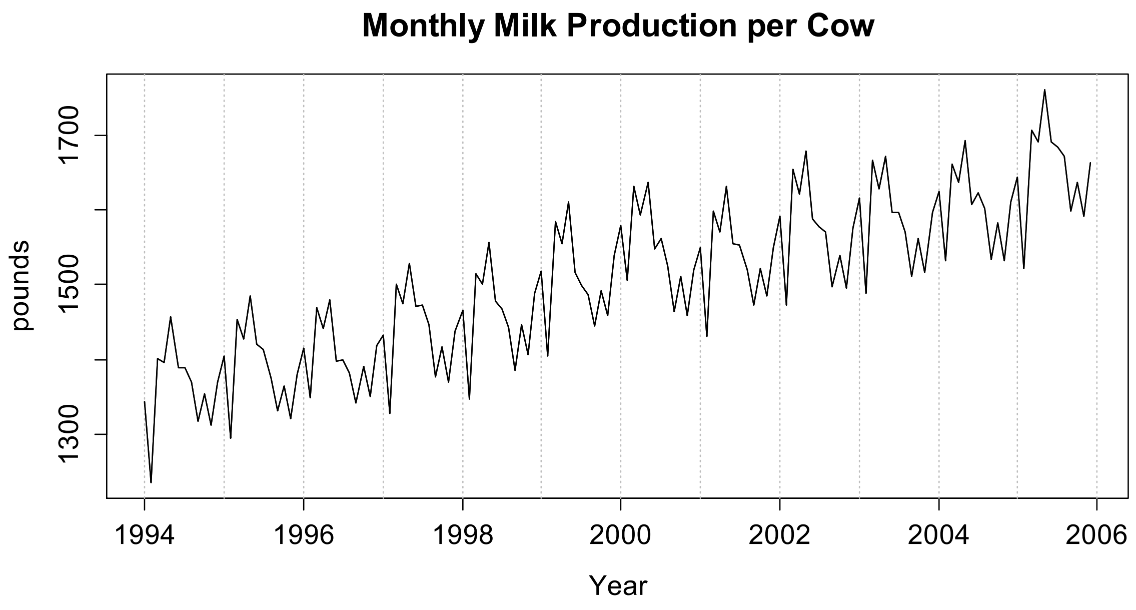
$$X_t = \frac{1}{n} \cdot \sum_{i=1}^{n} \sum_{j=1}^{k} C_{ij} \ ,$$

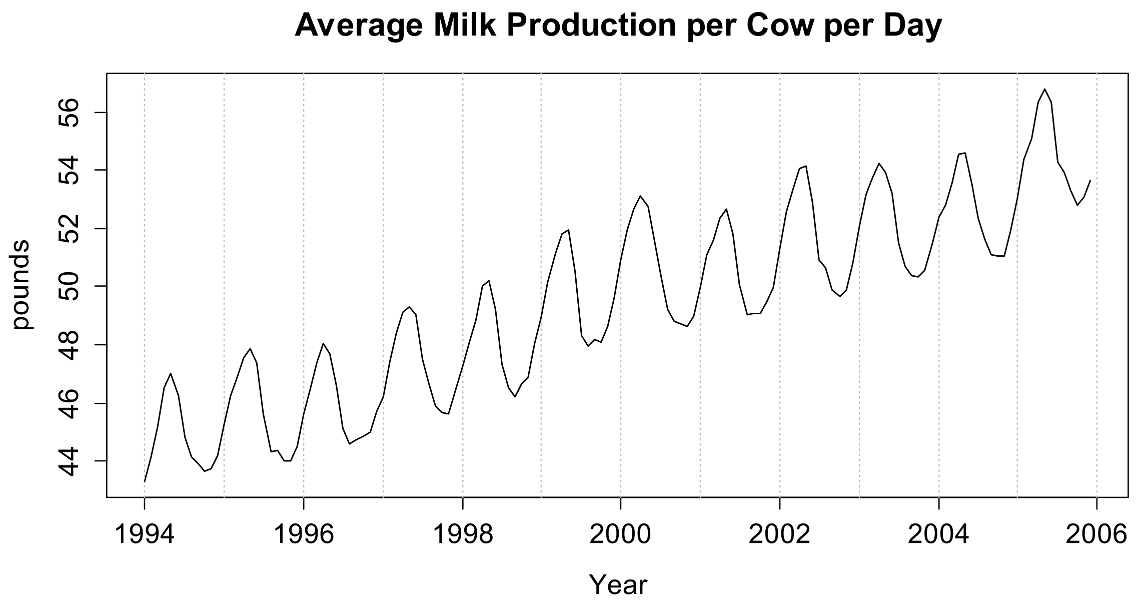
the average total milk production per cow in a month. In this formula, C_{ij} is the milk production of cow i on day j, n is the total number of cows observed and k is the number of days in month t. Obviously k ranges from 28 (for February) to 31 (e.g. January), adding undesired variation to X_t. It is much better to factor this out and consider the average *daily* milk production per cow in a month:

$$Y_t = \frac{1}{n} \cdot \sum_{i=1}^{n} \cdot \frac{1}{k} \sum_{j=1}^{k} C_{ij}$$

This simplifies the pattern in the data, see next page. Usually, this facilitates it to learn the signal, both for us humans and time series forecasting methods. Additionally, using daily averages solves the leap year problem that in every fourth year, February will have 29 rather than 28 days. While this affects the monthly total X_t, the daily average Y_t is unaffected. Please note that the `monthdays()` command from `library(TSA)` facilitates this standardization markedly.

```
> ## Loading the library
> library(TSA)
>
> ## Monthly totals
> plot(milk, xlab="Year", ylab="pounds", main="Monthly …")
> abline(v=1994:2006, col="grey", lty=3)
> lines(milk, lwd=1.5)
>
> ## Monthly average per day
> milk.adj <- milk/monthdays(milk)
> plot(milk.adj,xlab="Year",ylab="pounds", main="Average …")
> abline(v=1994:2006, col="grey", lty=3)
> lines(milk.adj, lwd=1.5)
```

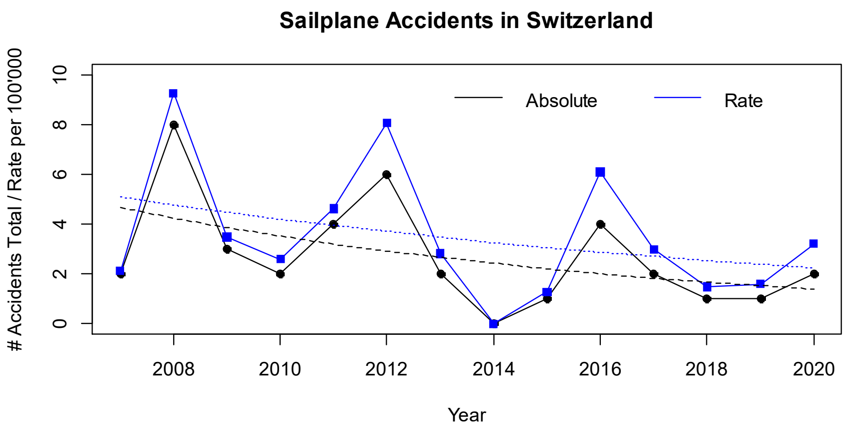




4.2.3 Population Adjustments

If time series data are affected by changes in the underlying population, it is much more insightful to show adjusted data. Let us consider the following yearly time series of fatal accidents with sailplanes in Switzerland from 2007 to 2020. The data were provided by STSB, the Swiss Transportation Safety Investigation Board. It is known to experts that the population and activity of sailplane pilots was declining over these years. Hence, a potential reduction in the number of accidents may not arise from better safety protocols, but from less exposure. It is hence much better to either provide per-capita data (i.e., the number of fatal accidents per active pilot). Even more precise is the accident rate, where the number of fatal accidents is standardized by the number of air traffic movements.

Alle Immatrikulationen. Alle Ops.	
Segelflugzeuge, Motorsegelflugzeuge	
Bewegungen für Flugabschnitte CH-CH (1)	Unfälle mit erheblich Verletzten oder Toten (2)
95126	2
86438	8
86438	3
77284	2
86632	4
74470	6
71063	2
79487	0
78269	1
65753	4
67117	2
67434	1
62890	1
62355	2



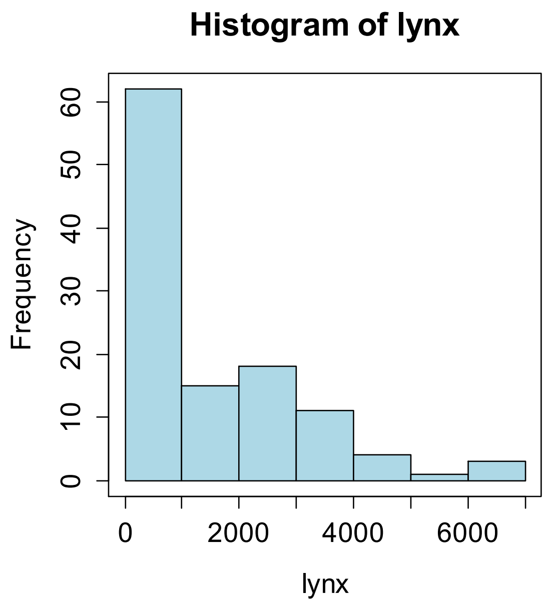
The difference between the two series is not extreme, but still substantial. The dashed/dotted lines correspond to an estimate of the expected accident number/rate. As it turns out, the decline in accident numbers is statistically significant, but the one for the rate is not (details omitted here, as the models used are too advanced at this point). It is a very general recommendation to show and analyze (time series) data in its most meaningful form – which means factoring out potential confounding or noise variables if possible.

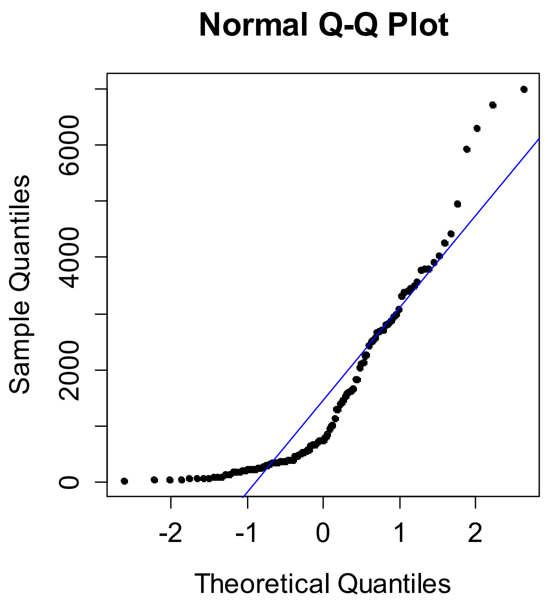
4.2.4 Log-Transformation

Many popular time series models and estimators (i.e. the usual ones for *mean*, *variance* and *correlation*) are based and most efficient in case of Gaussian distribution and additive, linear relations. However, data may exhibit different behavior. In such cases, we can often improve results by working with transformed values $g(x_1),...,g(x_n)$ rather than the original data $x_1,...,x_n$, The most popular and practically relevant transformation is $g(\cdot) = \log(\cdot)$. It is indicated if the variation in the series grows with the level, resp. if the series is on a relative scale where changes are better expressed in percent rather than in absolute values. This is another big advantage of the log-transformation: it is interpretable, i.e. the transformed values are the relative changes for the original values.

For time series where a log-transformation is beneficial, the marginal distribution is often (but not always!) right-skewed. Both properties are typical for time series which can take positive values only, such as the lynx trappings from section 1.2.2. It is easy to spot right-skewness by histograms and QQ-plots:

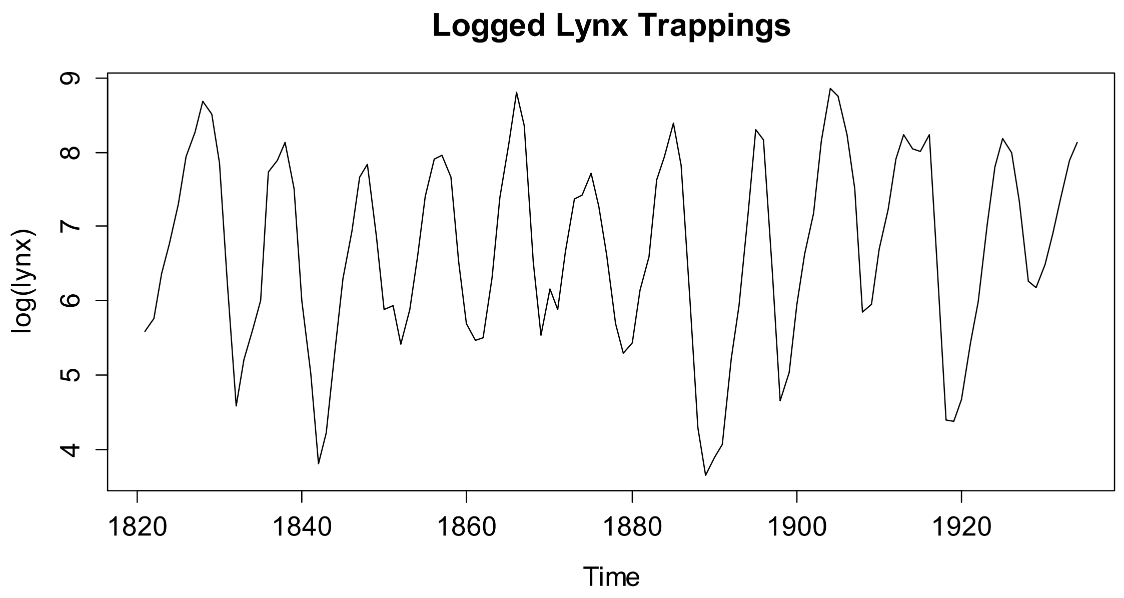
```
> hist(lynx, col="lightblue")
> qqnorm(lynx, pch=20); qqline(lynx, col="blue")
```



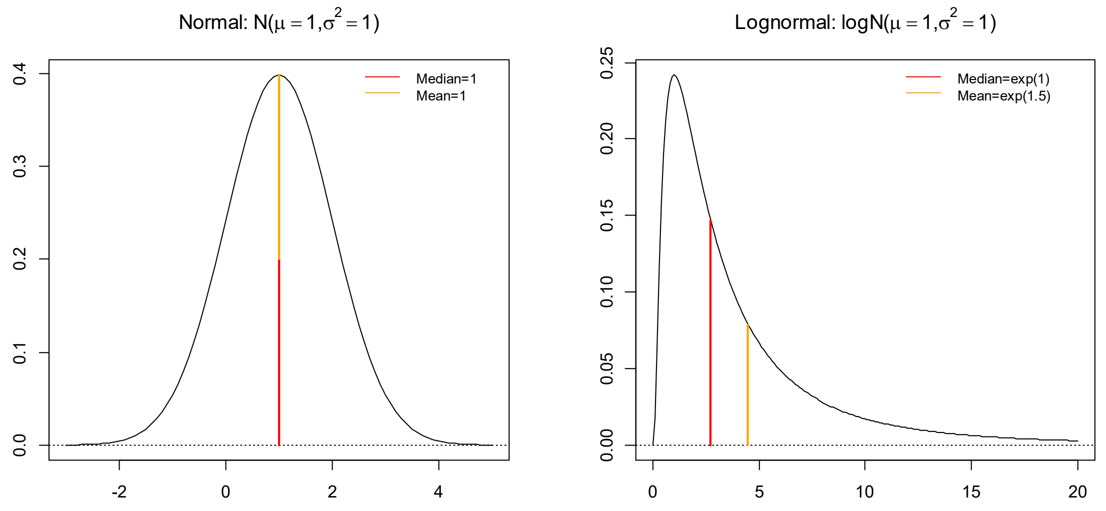


The lynx data are positive, on a relative scale and strongly right-skewed. Hence, a log-transformation proves beneficial. Implementing the transformation is easy in R:

```
> plot(log(lynx), main="Logged Lynx Trappings")
```



The data now follow a more symmetrical pattern; the extreme upward spikes are all gone. Another major advantage of the log-transformation is that model-based fitted values, forecasts and prediction intervals will not take negative values on the original scale. Often, this is a must for series which are strictly positive. However, an eye has to be had in the back-transformation to the original scale. For a symmetric distribution, e.g. the Gaussian that is appropriate on the log-transformed scale, mean and median take the same value. However, for the Lognormal distribution that we then have on the original scale, this is not the case.



While for a symmetric distribution, we can neglect the question about the representant for its location, we face a delicate decision in the right-skewed case. Mean and median differ, only one of the two is usually reported. What is the better choice? Unfortunately, there is no general answer (like when computing summary statistics for a skewed sample). What is certain is that when we use the simple $\exp(\cdot)$ for back-transforming to the original scale, the reported values are estimates for the median of the forecast distribution. That may be a valid choice, but there are applications where unbiased predictions are a must. In that case, a corrected back-transformation has to be applied. It is given by:

$$\exp(\hat{x}_t)\cdot\left(1+\frac{\hat{\sigma}_h^2}{2}\right), \text{ with } \hat{\sigma}_h^2 = \text{estimated } h\text{-step forecast variance}$$

Obviously, the bigger the forecast variance is, the more pronounced the difference between median and mean in the forecast distribution will be. Please note that all procedures from `library(forecast)` allow for convenient mean forecasting by setting the argument `biasadj=TRUE` when a log-transformation is involved, see the examples in the later chapters of this script. To conclude this chapter, we emphasize that either reporting median or mean are valid choices – but a decision for one of the two has to be made.

4.2.5 Box-Cox and Power Transformations

Another type of transformations sometimes used are power transformations which are of the form $g(x_t) = x_t^p$. The most popular instance is perhaps the square-root transformation with $p = 1/2$, which has some merit with count data. It's effect is similar to the one of the log-transformation, i.e. it stabilizes the variation of the series if that increases with the level. The drawback however is that the transformed values lack a direct interpretation – they are not relative changes as with the log, but just values on a different scale. The family of power transformations can be enhanced by the so-called Box-Cox transformation

$$g(x_t) = \frac{x_t^\lambda - 1}{\lambda} \text{ with } \lambda \neq 0.$$

Please note that the (non-allowed) case of $\lambda = 0$ corresponds to the (natural, i.e. base e) log-transformation discussed above. Again, Box-Cox transformed values lack a direct interpretation, but the method is of importance as many of the (to be presented) functions in `library(forecast)` allow for estimating λ. In fact, there is also the stand-alone function `BoxCox.lambda()` which allows for determining the most suitable transformation, i.e.:

```
> BoxCox.lambda(lynx)
[1] 0.1521849
```

The value turns out to be 0.15>0, indicating that a power transformation may be preferable to the log. On the other hand, we so lose the interpretability of the log, potentially without much practical benefit. We thus recommend favoring the log over a Box-Cox transformation for small λ's (i.e. smaller than $\approx \pm 0.3$). Likewise, we can often without any transformation at all if λ is estimated close to one.

If a Box-Cox transformation has been used, the issue of biased fitted values and point forecasts appears again. For obtaining the mean of the forecast distribution, a correction is needed:

$$(\lambda \hat{x}_t + 1)^{1/\lambda} \cdot \left(1 + \frac{\hat{\sigma}_h^2 (1-\lambda)}{2(\lambda \hat{x}_t + 1)^2} \right)$$

This formula looks quite complicated. Fortunately, if using the `forecast()` methods from `library(forecast)`, it is already implemented. Corrected, bias-free forecasts can be obtained by simply setting the argument `unbiased=TRUE`. We emphasize again that it is not obligatory to do so: in case it is not used, the point forecast will be the value where the realized value lies above resp. below with 50% probability each. If a corrected point forecast is given, it is the mean of all realized values – in case of a skewed distribution, this is not necessarily the better representant of what's going to happen.

4.2.6 Differencing for Time Series

While the logarithmic and Box-Cox transformations aim at stabilizing the dispersion within a time series, *differencing* is another form of transforming a non-stationary series with the aim of *removing trend* and/or *seasonal effects*.

Removing a Trend

We assume a time series X_t that is non-stationary in the mean, but free from a seasonal effect. The consequence is that $E[X] \neq const$. In that situation, taking first-order differences at lag 1 often produces a stationary time series:

$$Y_t = X_t - X_{t-1} \text{ with } E[Y_t] = const \approx 0 .$$

The practical interpretation of the differenced series Y_t is that now the changes in the data are monitored, but no longer the original values X_t itself. Usually, the differenced series Y_t is easier to analyze and model. In fact, the technique of first making the transformation to Y_t and the applying a time series model to it corresponds to the often-used and important $ARIMA$ class, presented in chapter 6. However, there are a few peculiarities about differencing. We consider the case where the trend function is exactly linear:

$$X_t = m_t + U_t \text{ with } m_t = \alpha + \beta t \text{ and } U_t \text{ is stationary with } E[U_t] = 0 .$$

This setup is also known as a *drift model*. The differenced series Y_t will be stationary, but it has mean $E[Y_t] = \beta \neq 0$. This is easily recognized from:

$$Y_t = X_t - X_{t-1} = (m_t + U_t) - (m_{t-1} + U_{t-1}) = \beta + (U_t - U_{t-1})$$

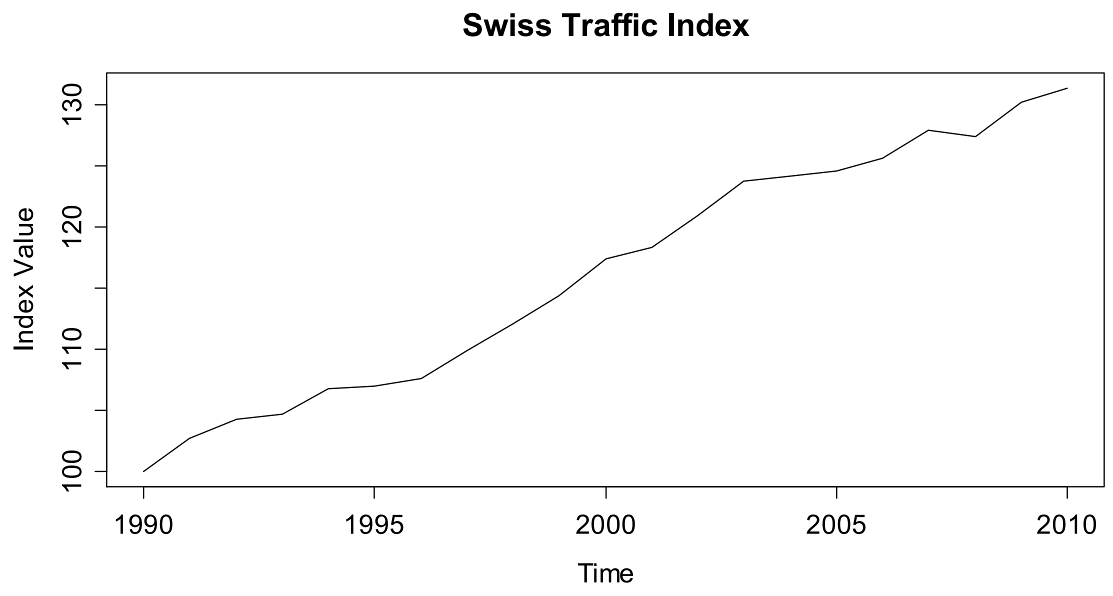
While the non-zero mean usually causes little sorrow, the creation of potentially new, artificial dependencies in the differenced series Y_t is more worrying. For illustration, we study the case where U_t is an *iid* random variable. Then:

$$\begin{aligned} Cov(Y_t, Y_{t-1}) &\approx Cov(U_t - U_{t-1}, U_{t-1} - U_{t-2}) \\ &= -Cov(U_{t-1}, U_{t-1}) \\ &\neq 0 \end{aligned}$$

This means that the differenced series Y_t is not *iid*, new dependencies were created. Still, trend removal by differencing can be very useful in practice. We illustrate by using a dataset that shows the traffic development on Swiss roads. The data are available from the federal road office (ASTRA) and show the indexed traffic amount from 1990-2010. We type in the values and plot the original series:

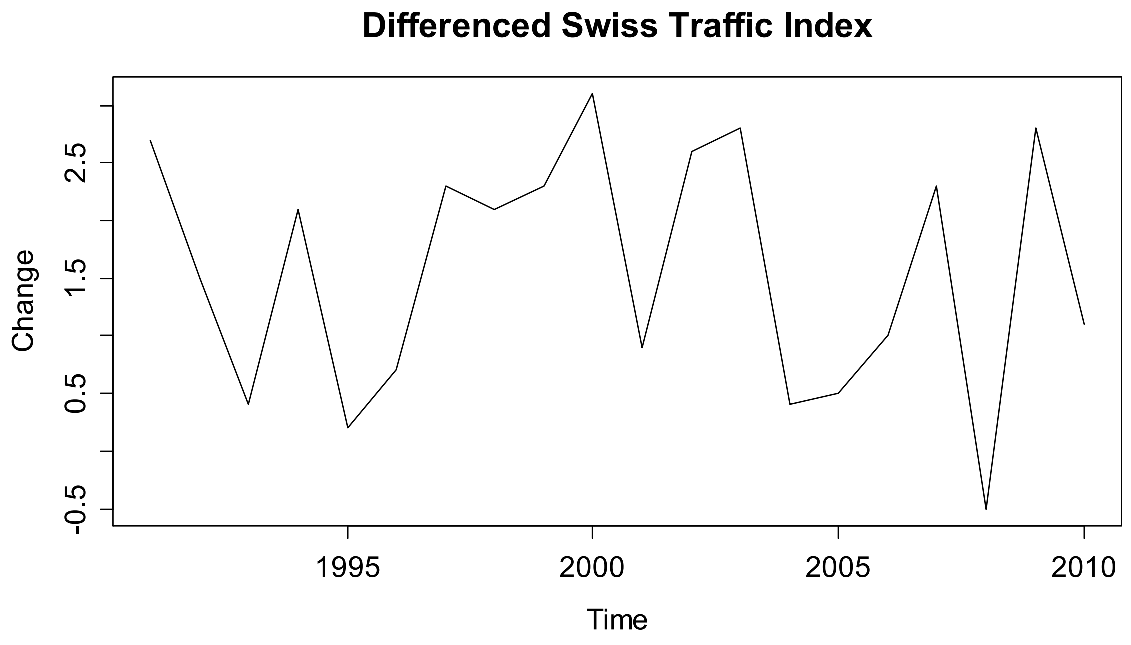
```
> SwissTraffic <- ts(c(100.0, 102.7, 104.2, 104.6, 106.7,
                       106.9, 107.6, 109.9, 112.0, 114.3,
                       117.4, 118.3, 120.9, 123.7, 124.1,
                       124.6, 125.6, 127.9, 127.4, 130.2,
                       131.3), start=1990, freq=1)
```

```
> plot(SwissTraffic)
```



There is a clear trend, which is close to linear, thus the simple approach should work well here. Taking first-order differences with lag 1 shows the yearly changes in the Swiss Traffic Index, which must now be a stationary series. In **R**, the job is done with function `diff()`.

```
> diff(SwissTraffic)
Time Series: Start = 1991
End = 2010
Frequency = 1
 [1]  2.7  1.5  0.4  2.1  0.2  0.7  2.3  2.1  2.3  3.1
[11]  0.9  2.6  2.8  0.4  0.5  1.0  2.3 -0.5  2.8  1.1
```



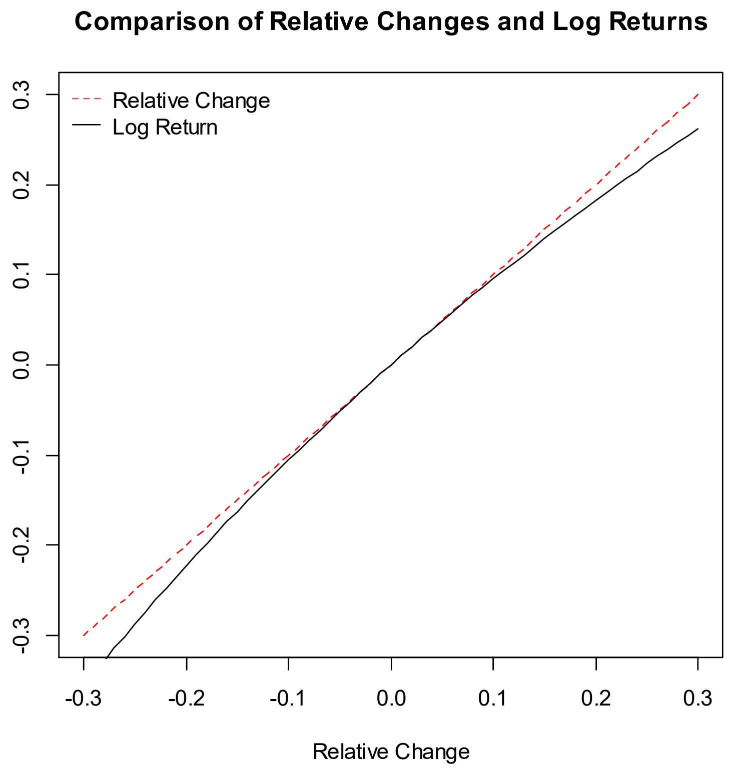
Please note that the time series of differences is now 1 instance shorter than the original series. The reason is that for the first year, 1990, there is no difference to the previous year available. The differenced series now seems to have a constant (but non-zero) mean, i.e. the trend was successfully removed.

Log-Transformation and Differencing

On a sidenote, we consider a series that was log-transformed first, before first-order differences with lag 1 were taken. An example is the SMI data that were shown in section 1.2.4. The result is the so-called *log return*, which is an approximation to the relative change, i.e. the percent in- or decrease with respect to the previous instance. In particular:

$$Y_t = \log(X_t) - \log(X_{t-1}) = \log\left(\frac{X_t}{X_{t-1}}\right) = \log\left(\frac{X_t - X_{t-1}}{X_{t-1}} + 1\right) \approx \frac{X_t - X_{t-1}}{X_{t-1}}$$

The approximation of the log return to the relative change is very good for small changes, and becomes a little less precise with larger values. For example, if we have a 0.00% relative change, then $Y_t = 0.00\%$, for 1.00% relative change we obtain $Y_t = 0.995\%$ and for 5.00%, $Y_t = 4.88\%$. The plot below illustrates the relation between the log returns Y_t and the simple returns resp. relative changes $(X_t - X_{t-1}) / X_t$.



We conclude with summarizing that for any non-stationary series which is also due to a log-transformation, the transformation is always carried out first, and then followed by the differencing!

The Backshift Operator

We here introduce the backshift operator B because it allows for convenient notation. When the operator B is applied to X_t it returns the instance at lag 1, i.e.

$$B(X_t) = X_{t-1}.$$

Less mathematically, we can also say that applying B means "go back one step", or "increment the time series index t by -1". The operation of taking first-order differences at lag 1 as above can be written using the backshift operator:

$$Y_t = (1-B)X_t = X_t - X_{t-1}$$

However, the main aim of the backshift operator is to deal with more complicated forms of differencing, as will be explained below.

Higher-Order Differencing

We have seen that taking first-order differences is able to reduce a linear trend in a time series to a constant. What has differencing to offer for polynomial trends, i.e. quadratic or cubic ones? We here demonstrate that it is possible to take higher order differences to remove also these, for example, in the case of a quadratic trend.

$$\begin{aligned} X_t &= \alpha + \beta_1 t + \beta_2 t^2 + U_t,\ U_t \ stationary \\ Y_t &= (1-B)^2 X_t \\ &= (X_t - X_{t-1}) - (X_{t-1} - X_{t-2}) \\ &= U_t - 2U_{t-1} + U_{t-2} + 2\beta_2 \end{aligned}$$

We see that the operator $(1-B)^2$ means that after taking "normal" differences, the resulting series is again differenced "normally". This is a discretized variant of taking the second derivative, and thus it is not surprising that it manages to remove a quadratic trend from the data. As we can see, Y_t is an additive combination of the stationary U_t's terms, and thus itself stationary. Again, if U_t was an independent process, that would clearly not hold for Y_t, thus taking higher-order differences (strongly!) alters the dependency structure.

Moreover, the extension to cubic trends and even higher orders d is straightforward. We just use the $(1-B)^d$ operator applied to series X_t. In **R**, we can employ function `diff()`, but have to provide argument `differences=d` for indicating the order of the difference d. In practice, we can use R function `ndiffs()` for determining the appropriate order of differencing d.

Removing Seasonal Effects by Differencing

For time series with monthly measurements, seasonal effects are very common. Using an appropriate form of differencing, it is possible to remove these, as well as potential trends. We take first-order differences with lag p:

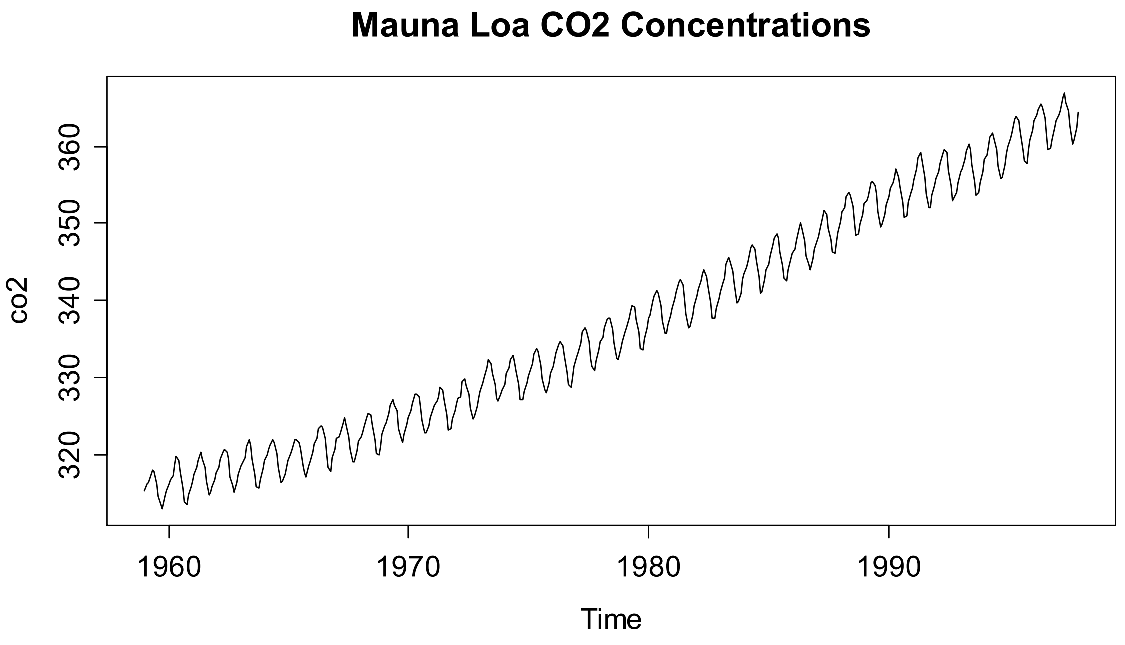
$$Y_t = (1-B^p)X_t = X_t - X_{t-p},$$

Here, p is the period of the seasonal effect, or in other words, the frequency of series, which is the number of measurements per time unit. The series Y_t then is made up of the changes compared to the previous period's value, e.g. the previous year's value. Also, from the definition, with the same argument as above, it is evident that not only the seasonal variation, but also a strictly linear trend will be removed. Usually, trends are not exactly linear. We have seen that taking differences at lag 1 removes slowly evolving (non-linear) trends well due to $m_t \approx m_{t-1}$. However, here the relevant quantities are m_t and m_{t-p}, and especially if the period p is long, some trend will usually be remaining in the data. Then, further action is required.

Example

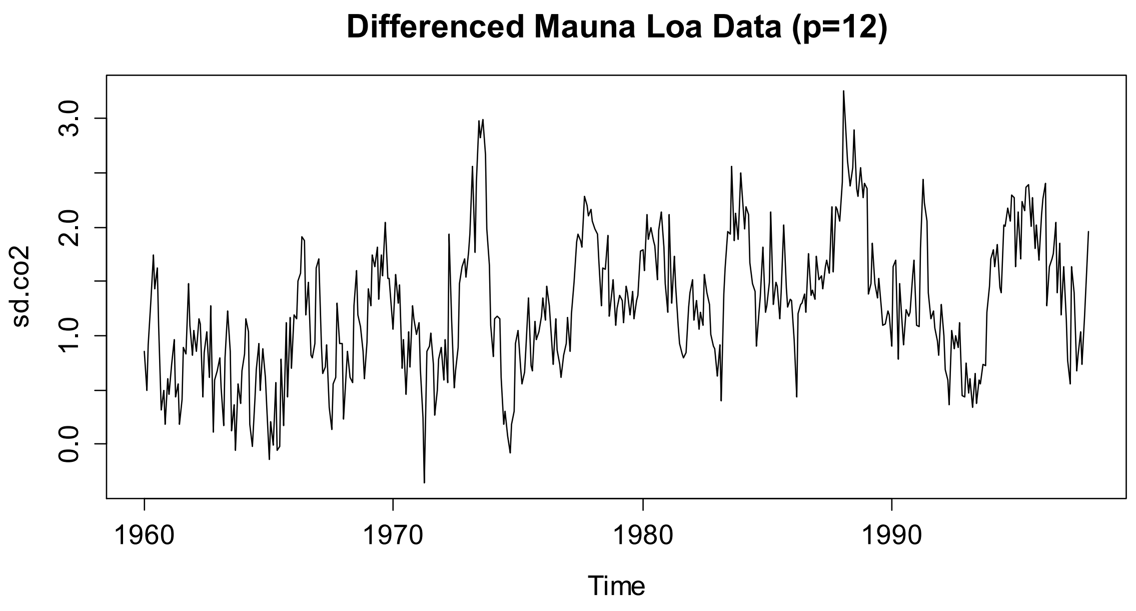
We are illustrating seasonal differencing using the Mauna Loa atmospheric CO_2 concentration data. This is a time series with monthly records from January 1959 to December 1997. It exhibits both a trend and a distinct seasonal pattern. We first load the data and establish a time series plot:

```
> data(co2)
> plot(co2, main="Mauna Loa CO2 Concentrations")
```



Seasonal differencing is very conveniently available in **R**. We use function `diff()`, but have to set argument `lag=...`. For the Mauna Loa data with monthly measurements, the correct lag is 12. This results in the series shown on the next page. Because we are comparing every record with the one from the previous year, the resulting series is 12 observations shorter than the original one. It is pretty obvious that some trend is remaining and thus, the result from seasonal differencing cannot be considered as stationary. As the seasonal effect is gone, we could try to add some first-order differencing at lag 1.

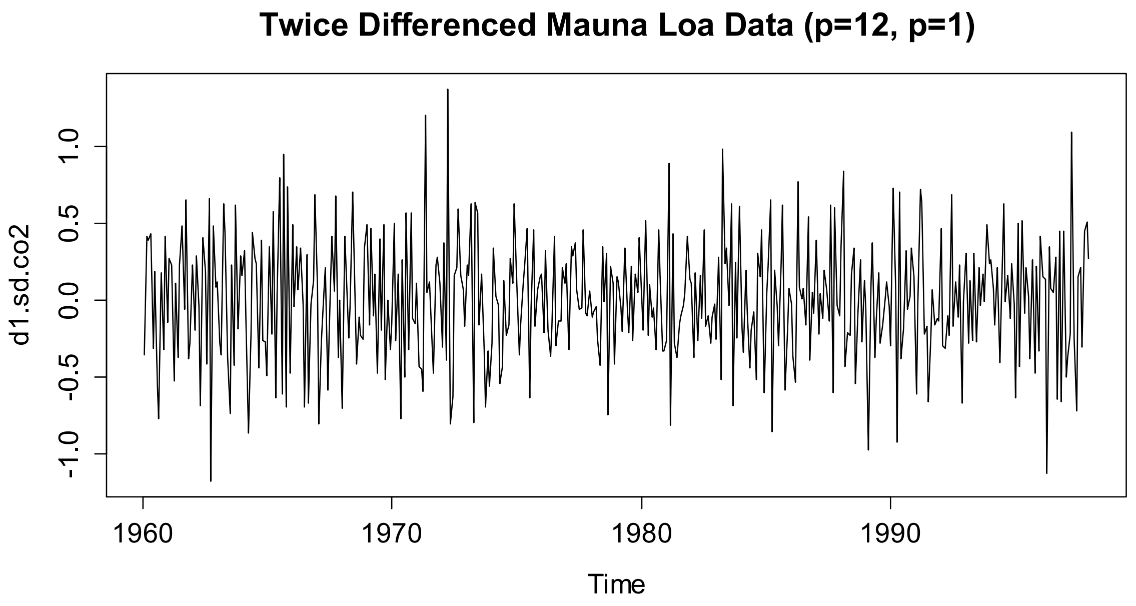
```
> sd.co2 <- diff(co2, lag=12)
> plot(sd.co2, main="Differenced Mauna Loa Data (p=12)")
```



The second differencing step indeed manages to produce a stationary series, as can be seen below. The equation for the final series is:

$$Z_t = (1-B)Y_t = (1-B)(1-B^{12})X_t .$$

The next step would be to analyze the autocorrelation of the series below and fit an $ARMA(p,q)$ model. Due to the two differencing steps, such constructs are also named $SARIMA$ models. They will be discussed in chapter 6.



We conclude this section by emphasizing that differencing is quick and simple, and (correctly done) manages to remove any trend and/or seasonality. The resulting series are “changes” between observations, but can be difficult to characterize beyond that. Moreover, the resulting series are shorter than the originals.

4.3 Decomposition of Time Series

4.3.1 The Basics

We have learned in section 2.2 that stationarity is an important prerequisite for being able to statistically learn from time series data. However, many of the example series exhibit either trend and/or seasonal effect, and thus are non-stationary. In this section, we will learn how to deal with that. First, we need to define what trend and seasonality mean.

Trend

A deterministic trend in a time series is a long-term change in the mean, induced by external factors. Typical examples among the series we have seen so far include the *Air Passenger*, *Australian Production*, *Maine Unemployment* and *SMI* data. The Lynx data on the other hand are considered to arise from a stationary process, hence are without trend. The obvious oscillation is attributed to a random cyclic component.

Seasonal Effect

A seasonal component is a deterministic cyclic component in a time series with a fixed and known frequency, often caused by the way the measurements are obtained. The most typical case is the seasonal effect in monthly data where the measurement period comprises of multiple years, this coined the term. Typical examples include the *Air Passenger*, *Australian Production* and *Maine Unemployment* data. Seasonal components can also be present in e.g. hourly data that were observed over several days, it is a "daily pattern" in this case. On the other hand, the Lynx data are (as most yearly data) non-seasonal. They do have a random cyclic component, but it is not a seasonal effect!

Decomposition Model

The standard model describes a time series X_t as an *additive composition* of a (potentially absent) deterministic *trend* component m_t, a (potentially absent) *seasonal effect* s_t and a *stationary remainder* term R_t. Hence,

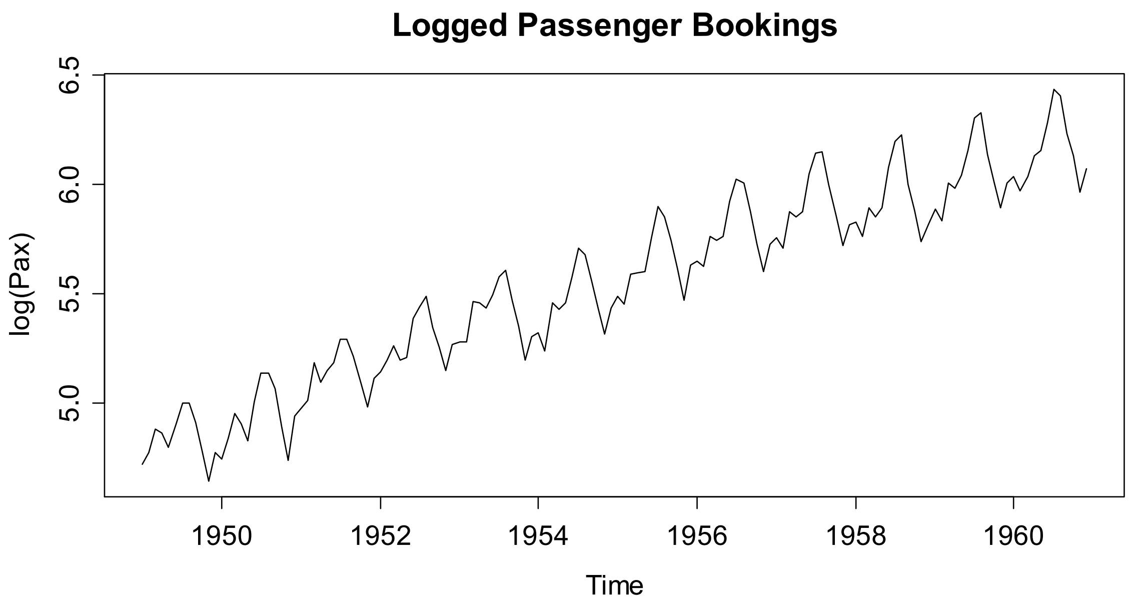
$$X_t = m_t + s_t + R_t ,$$

where X_t is the time series process at time t, m_t is the trend, s_t is the seasonal effect, and R_t is the remainder, i.e. a sequence of usually correlated random variables with mean zero. In practice however, many time series exhibit an increase in seasonal and random variation with the level of the series. This is the case in all seasonal series presented in this script, i.e. *Air Passenger*, *Australian Production* and *Maine Unemployment* data. For making the additive decomposition model a valid choice, the data need to be transformed with either a Box-Cox resp. power transformation, or much more often, the logarithm. The aim is to stabilize the seasonal and remainder variation to constant magnitude, so that the additive decomposition approach applies. Simple math demonstrates that a multiplicative composition of a series on its original scale becomes additive after a logarithmic transformation:

$$\log(X_t) = \log(m_t \cdot s_t \cdot R_t) = \log(m_t) + \log(s_t) + \log(R_t) = m'_t + s'_t + R'_t$$

For illustration, we carry out a log-transformation on the air passenger bookings:

```
> plot(log(AirPassengers), ylab="log(Pax)", main=...)
```

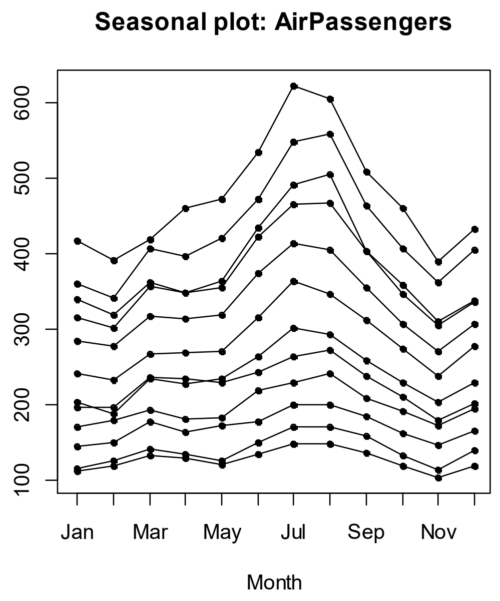


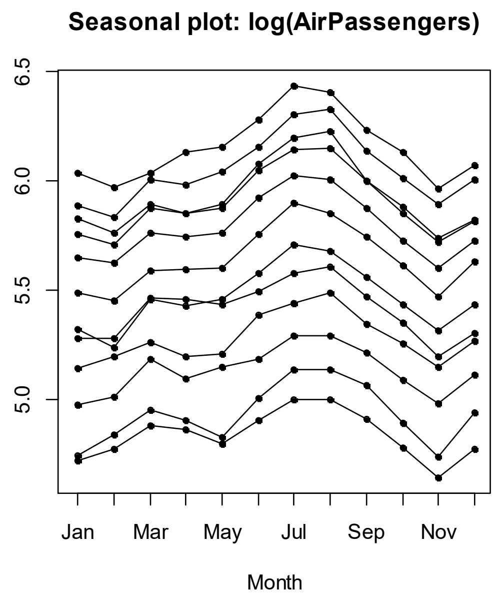
The plot shows that indeed, the magnitude of seasonal effect and random variation now seem to be less dependent of the level of the series than it was initially. Thus, the multiplicative model is much more appropriate for the Air Passenger data than the additive one. Alternatively, we could also estimate a Box-Cox transformation:

```
> BoxCox.lambda(AirPassengers)
[1] -0.2947156
```

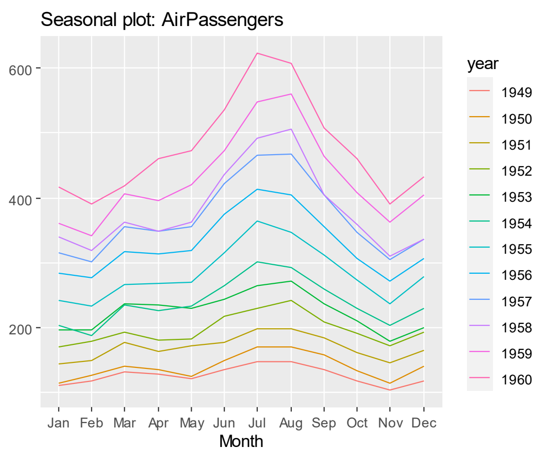
The value is quite close to zero so that we prefer to work with the easier-to-interpret logarithmic transformation. Please note that if using any other Box-Cox transformation than the logarithm, an additive decomposition would be estimated on the transformed scale, but the original data do then not follow a multiplicative decomposition model. Besides the time series plot of original and transformed data and the `BoxCox.lambda()` value, further evidence for a transformation can be found in the seasonal plots. Very helpful are the R functions `seasonplot()` or its more modern counterpart `ggseasonplot()`.

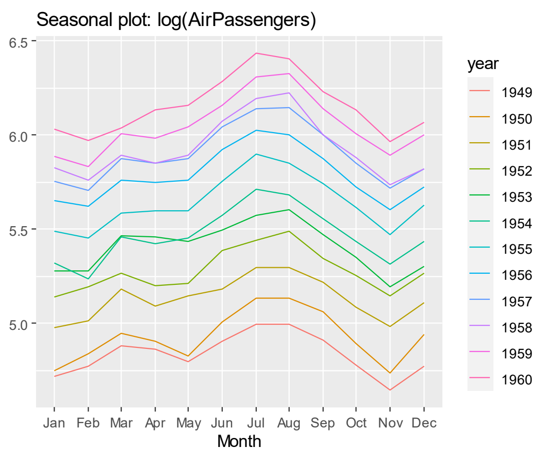
```
> seasonplot(AirPassengers, pch=20)
> seasonplot(log(AirPassengers), pch=20)
```





```
> ggseasonplot(AirPassengers)
```





The left one on the untransformed data clearly shows that the difference between summer and winter is larger in the later years when the passenger figures are higher. After the log-transformation, the magnitude of the seasonal differences are more or less constant, though. However, a further snag is that the seasonal effect seems to alter over time rather than being constant. In earlier years, a prominent secondary peak in March is apparent. Over time, this erodes away, but on the other hand, the summer peak seems to be ever rising. The issue of how to deal with evolving seasonal effects will be addressed later in chapter 4.3.3.

4.3.2 Smoothing, Filtering

Our next goal is to define a decomposition procedure that yields explicit trend, seasonality and remainder estimates $\hat{m}_t$, $\hat{s}_t$ and $\hat{R}_t$. In the absence of a seasonal effect, the trend of a time series can simply be obtained by applying an *additive linear filter*:

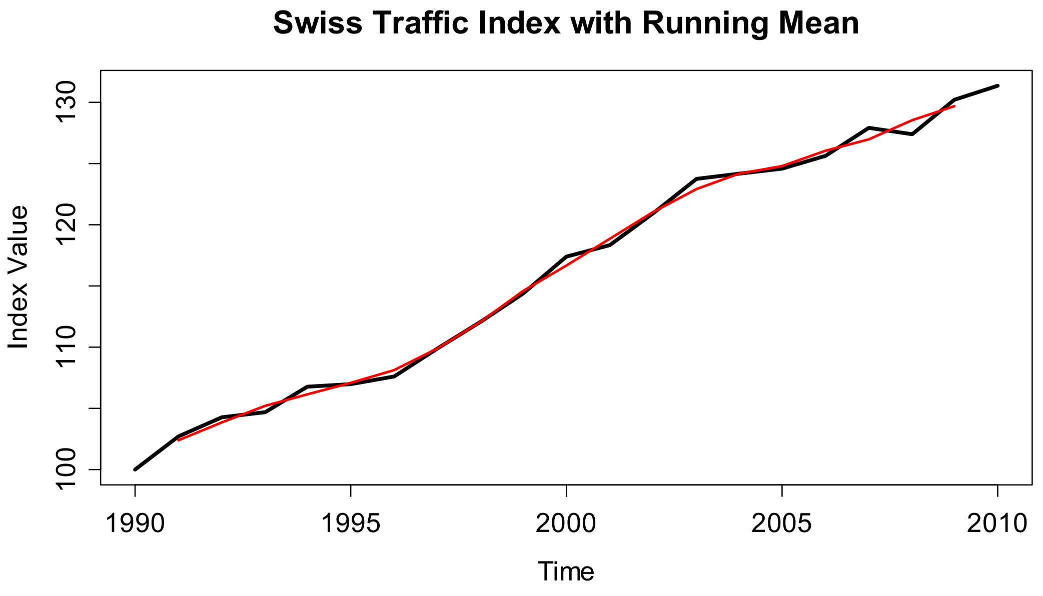
$$\hat{m}_t = \sum_{i=-p}^{q} a_i X_{t+i}$$

This definition is general, it allows for arbitrary weights and asymmetric windows. The most popular implementation is with $p = q$ and $a_i = 1/(2p+1)$, i.e. a *running mean* or *moving average estimator* with symmetric window and uniformly distributed weights. The window size is the smoothing parameter.

Example: Trend Estimation with Running Mean

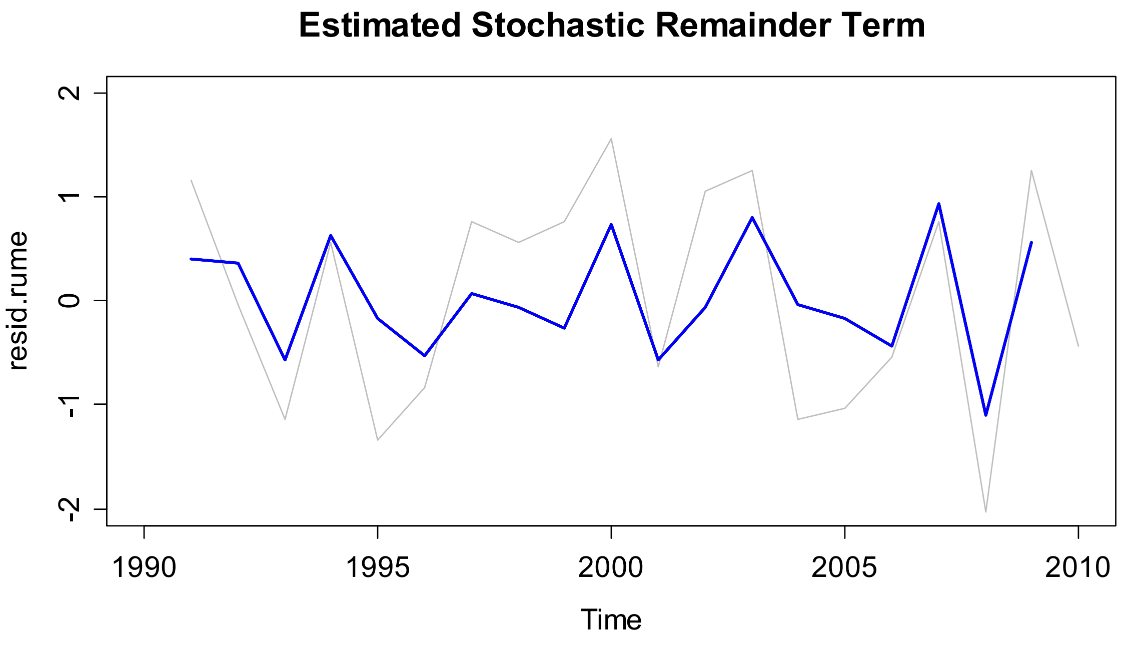
We here again consider the Swiss Traffic data that were already exhibited before. They show the indexed traffic development in Switzerland between 1990 and 2010. Linear filtering is available with function `filter()` from the base functionality in **R**., whereas for moving average computation, function `ma()` from `library(forecast)` is even more convenient.

```
> trend.est <- filter(SwissTraffic, filter=c(1,1,1)/3)
> trend.est <- ma(SwissTraffic, order=3)
```



```
> trend.est
Time Series: Start = 1990, End = 2010, Frequency = 1
 [1]       NA 102.3000 103.8333 105.1667 106.0667 107.0667
 [7] 108.1333 109.8333 112.0667 114.5667 116.6667 118.8667
[13] 120.9667 122.9000 124.1333 124.7667 126.0333 126.9667
[19] 128.5000 129.6333       NA
```

In our example, we chose the trend estimate to be the mean over three consecutive observations, resp. a 3-year moving average. This has the consequence that for both the first and the last instance of the time series, no trend estimate is available. We will later present more sophisticated methods that also allow for estimates near the endpoints. Furthermore, it is apparent that the Swiss Traffic series has a very strong trend signal, whereas the remaining stochastic term is comparably small in magnitude. We can now compare the estimated remainder term from the running mean trend estimation to the result from differencing:



The blue line is the remainder estimate from running mean approach, while the grey one resulted from differencing with lag 1. We observe that the latter has bigger variance; and, while there are some similarities between the two series, there are also some prominent differences – please note that while both seem stationary, they are different.

Trend Estimation for Seasonal Data

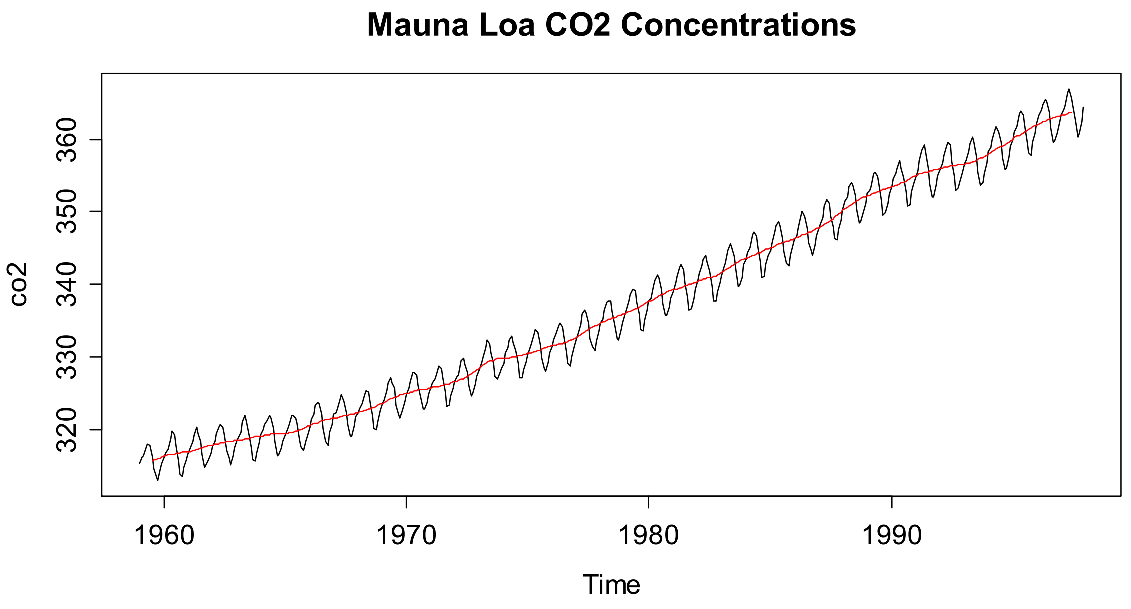
We now turn our attention to time series that show both trend *and* seasonal effect. The goal is to specify a filtering approach that allows trend estimation for periodic data. We still base this on the running mean idea, but have to make sure that we average over a full period. For monthly data, the formula is:

$$\hat{m}_t = \frac{1}{12}\left(\frac{1}{2}X_{t-6} + X_{t-5} + \ldots + X_{t+5} + \frac{1}{2}X_{t+6}\right), \text{ for } t = 7,\ldots,n-6$$

Be careful, as there is a slight snag if the frequency is even: if we estimate the trend for December, we use data from July to May, and then also add half of the value of the previous June, as well as half of the next June. This is required for having a window that is centered at the time we wish to estimate the trend. Using **R**'s function `filter()`, with appropriate choice of weights, we can compute the seasonal running mean. Or we can use function `ma()` with argument `order=12` for the same task. We illustrate this with the Mauna Loa CO_2 data.

```
> wghts     <- c(.5,rep(1,11),.5)/12
> trend.est <- filter(co2, filter=wghts, sides=2)
> trend.est <- ma(co2, order=12, centre=TRUE)
> plot(co2, main="Mauna Loa CO2 Concentrations")
> lines(trend.est, col="red")
```

We obtain a trend which fits well to the data. It is not a linear trend, rather it seems to be slightly progressively increasing, and it has a few kinks, too.



We finish this section about trend estimation using linear filters by stating that other smoothing approaches, e.g. *running median estimation*, the *loess smoother* and many more are valid choices for trend estimation, too. In fact, several of them have clear advantages over simple movering average approaches.

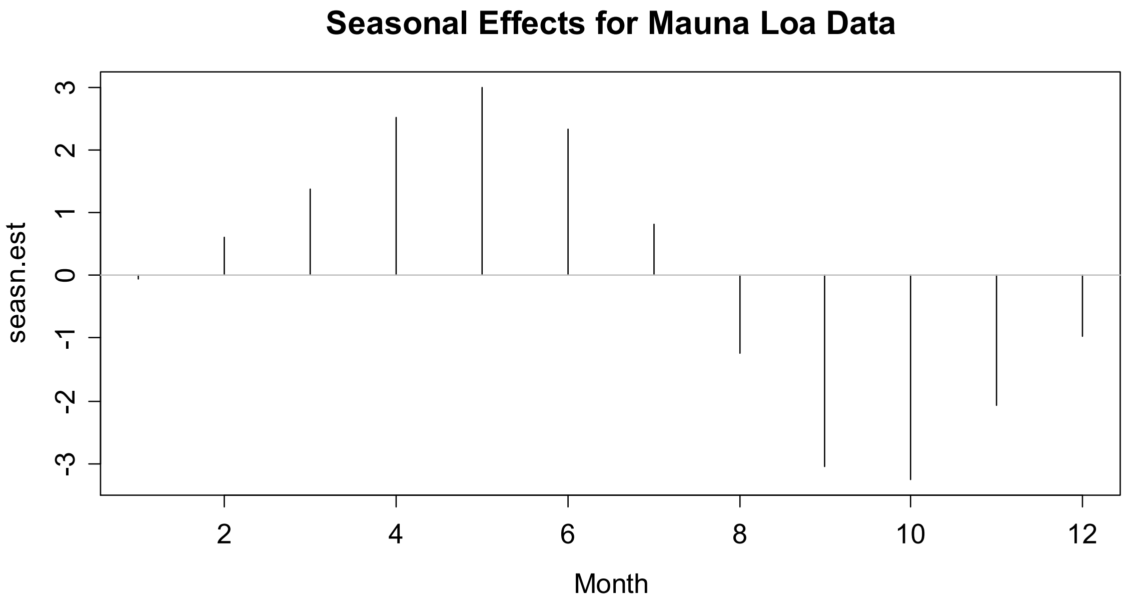
Estimation of the Seasonal Effect

For fully decomposing periodic series such as the Mauna Loa data, we also need to estimate the seasonal effect. This is done on the basis of the trend adjusted data: simple averages over all observations from the same seasonal entity are taken. The following formula shows the January effect estimation for the Mauna Loa data, a monthly series which starts in January and has 39 years of data.

$$\hat{s}_{Jan} = \hat{s}_1 = \hat{s}_{13} = ... = \frac{1}{39} \cdot \sum_{j=0}^{38} (x_{12j+1} - \hat{m}_{12j+1})$$

In **R**, a convenient way of estimating such seasonal effects is by generating a factor for the months, and then using the `tapply()` function. Please note that the seasonal running mean naturally generates NA values at the start and end of the series, which need be removed in the seasonal averaging process.

```
> trend.adj <- co2-trend.est
> month     <- factor(rep(1:12,39))
> seasn.est <- tapply(trend.adj, month, mean, na.rm=TRUE)
> plot(seasn.est, type="h", xlab="Month")
> title("Seasonal Effects for Mauna Loa Data")
> abline(h=0, col="grey")
```

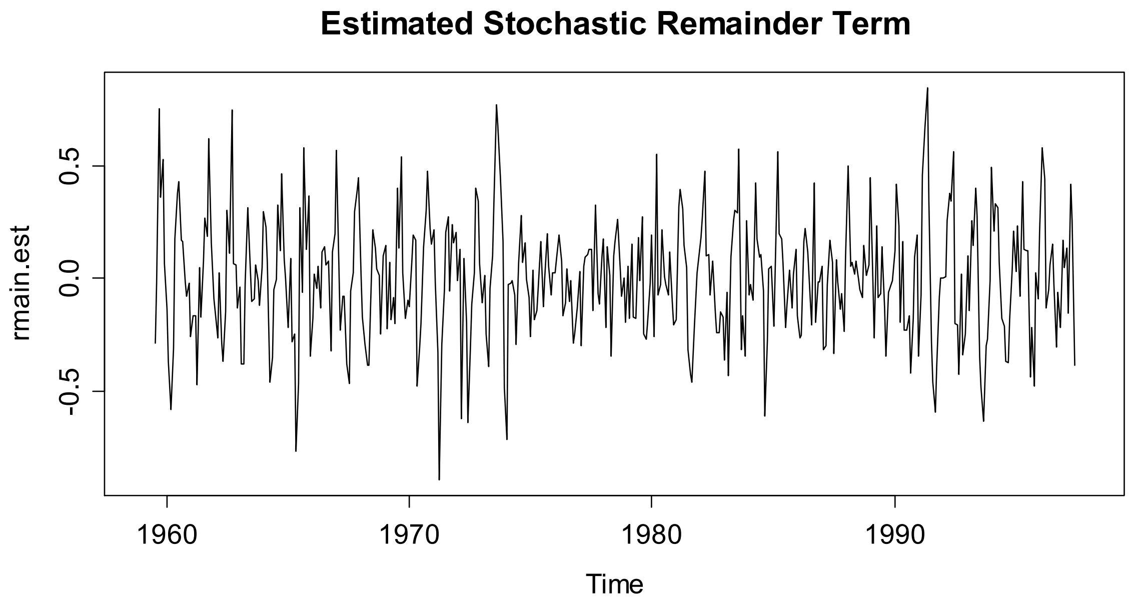


In the plot above, we observe that during a period, the highest values are usually observed in May, whereas the seasonal low is in October. The estimate for the remainder at time t is simply obtained by subtracting estimated trend and seasonality from the observed value.

$$\hat{R}_t = x_t - \hat{m}_t - \hat{s}_t$$

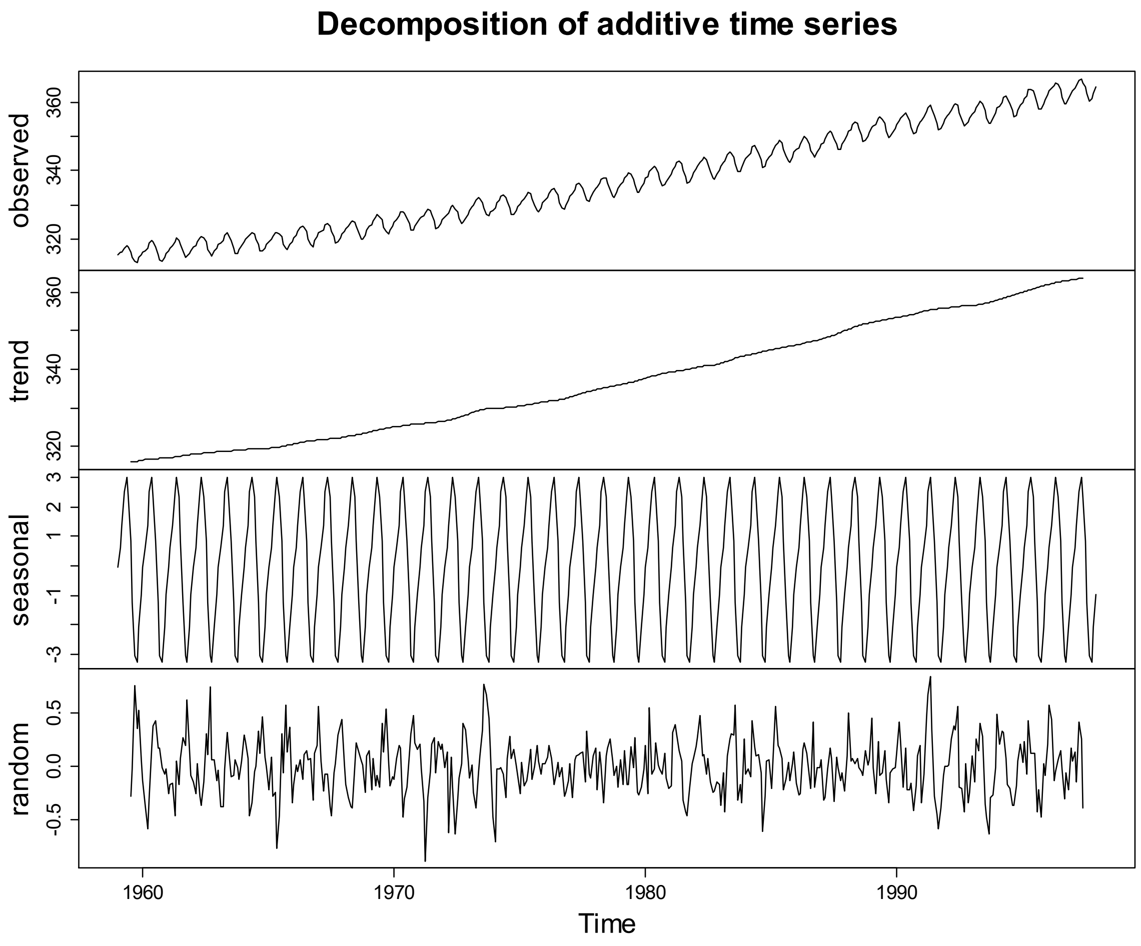
From the plot on the next page, it seems as if the estimated remainder still has some periodicity and thus it is questionable whether it is stationary. The periodicity is due to the fact that the seasonal effect is not constant but slowly evolving over time. In the beginning, we tend to overestimate it for most months, whereas in the end, we underestimate. We will address the issue on how to visualize evolving seasonality below in section 4.3.3 about STL-decomposition. A further option for dealing with non-constant seasonality is given by the exponential smoothing approach which is covered in chapter 8.

```
> rmain.est <- co2-trend.est-rep(seasn.est,39)
> plot(rmain.est, main="Estimated Stochastic Remainder Term")
```



Moreover, we would like to emphasize that R offers the convenient `decompose()` function for running mean estimation and seasonal averaging.

```
> co2.dec <- decompose(co2)
> plot(co2.dec)
```



Please note that `decompose()` only works with periodic series where at least two full periods were observed; else it is not mathematically feasible to estimate trend and seasonality from a series. The `decompose()` function also offers the neat plotting method shown above that generates the four frames above with the series, and the estimated trend, seasonality and remainder. Except for the different visualization, the results are exactly the same as what we had produced with our do-it-yourself approach.

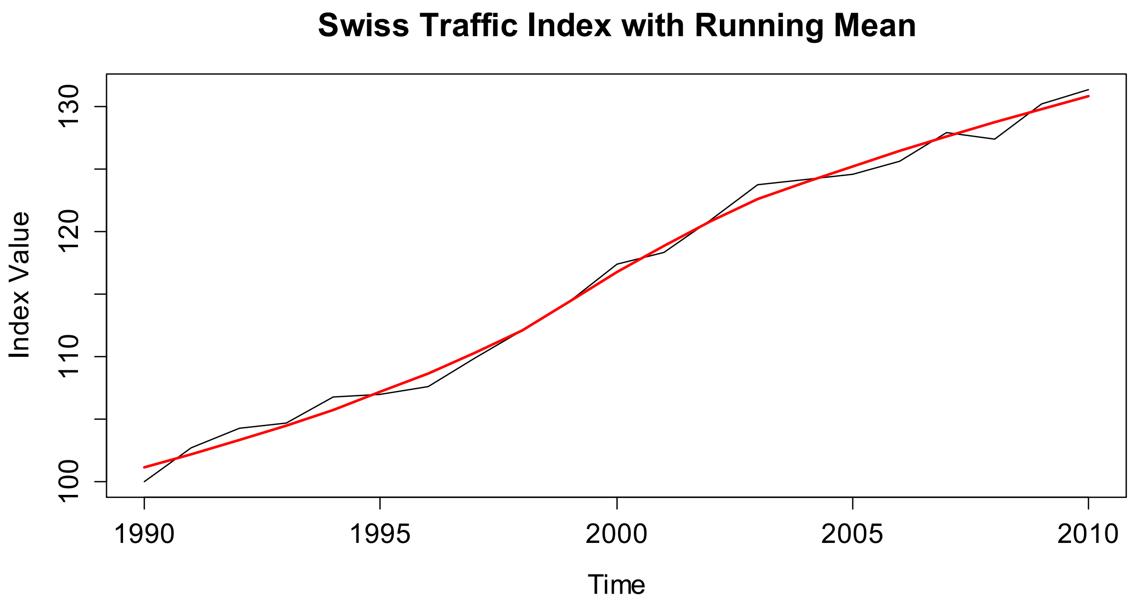
4.3.3 Seasonal-Trend Decomposition with LOESS

It is well known that the running mean resp. moving average is not the best smoother around. Thus, potential for improvement exists. While there is a dedicated R procedure for decomposing periodic series into trend, seasonal effect and remainder, we have to do some handwork in non-periodic cases.

Trend Estimation with LOESS

We here again consider the Swiss Traffic dataset, for which the trend had already been estimated above. Our goal is to re-estimate the trend with *LOESS*, a smoothing procedure that is based on local, weighted regression. The aim of the weighting scheme is to reduce potentially disturbing influence of outliers. Applying the LOESS smoother with (the often optimal) default settings is straightforward:

```
> fit   <- loess(SwissTraffic~time(SwissTraffic))
> trend <- predict(fit)
```



We observe that the estimated trend, in contrast to the running mean result, is now smooth and allows for interpolation within the observed time. Also, the `loess()` algorithm returns trend estimates which extend to the boundaries of the dataset. In summary, we recommend to always perform trend estimation with LOESS.