Lecture Notes Business Analytics II

Financial Market Analysis in R

Phillipp Gnan* Florian Pauer†

December 2, 2021

*WU (Vienna University of Economics and Business), Department of Finance, Accounting and Statistics, Welthandelsplatz 1, 1020 Vienna, Austria; *email:* phillipp.gnan@wu.ac.at

†WU (Vienna University of Economics and Business), Department of Finance, Accounting and Statistics, Welthandelsplatz 1, 1020 Vienna, Austria; *email:* florian.pauer@wu.ac.at

Contents

1. Loading data **3**
- 1.1. `.csv` files . . . 3
- 1.2. `.RData` files . . . 4
- 1.3. Cleaning data . . . 4

2. Basic cross-sectional data visualization **9**
- 2.1. Bar plots . . . 9
- 2.2. Histogram . . . 10
- 2.3. Box plot . . . 13
- 2.4. Scatter plot . . . 16
- 2.5. QQ-plot . . . 18

3. The `xts` package **22**
- 3.1. Converting to an `xts` object . . . 22
- 3.2. Plotting with `xts` . . . 23
- 3.3. Handling `xts` objects . . . 28
  - 3.3.1. Timeindex, subsetting time, lags and differences . . . 28
  - 3.3.2. ATTENTION!!! . . . 33
  - 3.3.3. `apply.weekly()` . . . 36

4. From stocks to portfolios **41**
- 4.1. Aggregating returns . . . 41
- 4.2. Sorting stocks . . . 48
  - 4.2.1. `R` functions `sort`, `rank`, `order` and `cut` . . . 48
  - 4.2.2. Sorting stocks repeatedly over time . . . 51

5. Hypothesis Tests **55**
- 5.1. Nonparametric Hypothesis Tests . . . 55
- 5.2. Parametric Hypothesis Tests . . . 57

6. Measuring Risk **61**
- 6.1. Volatility . . . 61
- 6.2. Covariance and correlation . . . 62
- 6.3. Skewness . . . 65
- 6.4. Value at Risk and Expected Shortfall . . . 66
- 6.5. Sharpe Ratio . . . 67

7. Numerical Optimization **69**

8. Linear Regression Analysis **76**
- 8.1. Univariate Linear Regression . . . 76
- 8.2. Multivariate Linear Regression . . . 77
- 8.3. Additional Example: Fama & French Three Factor Model . . . 89

9. PCA – Application to the German sovereign yield curve **92**
- 9.1. Overview of the data and the problem . . . 92
- 9.2. PCA – intuition and implementation in `R` . . . 96

A. Modern Portfolio Theory **109**
A.1. Simplified Model 109
A.2. Multiple Assets Model 110

B. OLS Estimates **114**
B.1. Univariate OLS 114

C. Principal Component Analysis - a geometric explanation **115**

1. Loading data

In this section, you will learn the most essential tools for loading data into an `R` environment. In this course we will provide you either with .csv or .RData files which are introduced in the following.

1.1. `.csv` files

A common way to store data that is easily representable in "table form" – i.e. all observations have in priciple identical dimensions – is the comma seperated values or csv format. `.csv` files store one observation per line using a seperator, usually ",", ";", or " " to seperate the different dimensions or fields observed. The `R` package `utils` provides a function to read `.csv` files into the workspace called `read.csv`. The first and most important anrgument to this function is the path where the `.csv` file is stored: `"yourpath/yourdata.csv"`. Note that R uses "/" in paths, while MS Windows uses "\". Before being able to read the file correctly you first have to figure out which seperator is used in the file in question. Using the `sep=""` argument you tell the function which seperator to use. In principle this is sufficient to read the file, but R will recognize all data fields as string by default.

```
german.interest <- read.csv("../data/germany.csv", sep=";")
head(german.interest)
```

```
##         Date     X3M     X6M     X1Y     X2Y     X3Y     X4Y     X5Y     X6Y
## 1 11.12.2000 4,91491 4,90332 4,74611 4,65575 4,64359 4,65474 4,67264 4,75868
## 2 12.12.2000 4,91491 4,90622 4,74963 4,65641  4,6453 4,65257 4,67391 4,76458
## 3 13.12.2000 4,91395 4,90332 4,70384 4,59706 4,58216 4,58877 4,60987   4,701
## 4 14.12.2000 4,90139 4,87723 4,68483 4,52651  4,5504 4,55694 4,57871 4,67291
## 5 15.12.2000 4,89752 4,87337 4,66248 4,50157 4,51175 4,50756  4,5284 4,62212
## 6 18.12.2000 4,88206 4,85597 4,61884 4,47767 4,47037 4,46584 4,48716 4,58597
##       X7Y     X8Y     X9Y    X10Y    X15Y    X20Y    X30Y
## 1 4,81739 4,84323 4,83299 4,85954 5,05866 5,31231 5,58972
## 2  4,8291 4,85977 4,85315 4,88013 5,07307 5,32375  5,5981
## 3 4,76851 4,80935 4,80022 4,83391   5,023 5,27704 5,55546
## 4 4,73771 4,78923  4,7809 4,81444 4,99204 5,22764 5,48557
## 5 4,69189 4,74841 4,74071 4,77875  4,9554 5,19238 5,45185
## 6 4,65824 4,72371 4,72434 4,75643 4,94766 5,19411 5,46294
```

```
german.interest$X3M[1]+german.interest$X6M[1]
```

```
## Error in german.interest$X3M[1] + german.interest$X6M[1]:  nicht-numerisches
Argument für binären Operator
```

Since all fields are interpreted as strings you need to convert the data to the respective format. Given the example above we would like the first column to be of type `Date` and all other columns of type `numeric`. Now, if the decimal point in the `.csv` file would indeed be a point, `R` would recognize the field as number by itself. As this is not the case we first need to tell the function which decimal seperator is used in the file.

```
german.interest <- read.csv("../data/germany.csv", sep=";", dec=",")
head(german.interest)

##         Date     X3M     X6M     X1Y     X2Y     X3Y     X4Y     X5Y     X6Y
## 1 11.12.2000 4.91491 4.90332 4.74611 4.65575 4.64359 4.65474 4.67264 4.75868
## 2 12.12.2000 4.91491 4.90622 4.74963 4.65641 4.64530 4.65257 4.67391 4.76458
## 3 13.12.2000 4.91395 4.90332 4.70384 4.59706 4.58216 4.58877 4.60987 4.70100
## 4 14.12.2000 4.90139 4.87723 4.68483 4.52651 4.55040 4.55694 4.57871 4.67291
## 5 15.12.2000 4.89752 4.87337 4.66248 4.50157 4.51175 4.50756 4.52840 4.62212
## 6 18.12.2000 4.88206 4.85597 4.61884 4.47767 4.47037 4.46584 4.48716 4.58597
##       X7Y     X8Y     X9Y    X10Y    X15Y    X20Y    X30Y
## 1 4.81739 4.84323 4.83299 4.85954 5.05866 5.31231 5.58972
## 2 4.82910 4.85977 4.85315 4.88013 5.07307 5.32375 5.59810
## 3 4.76851 4.80935 4.80022 4.83391 5.02300 5.27704 5.55546
## 4 4.73771 4.78923 4.78090 4.81444 4.99204 5.22764 5.48557
## 5 4.69189 4.74841 4.74071 4.77875 4.95540 5.19238 5.45185
## 6 4.65824 4.72371 4.72434 4.75643 4.94766 5.19411 5.46294

german.interest$X3M[1]+german.interest$X6M[1]

## [1] 9.81823
```

Sometimes it is necessary to edit your `.csv` file beforehand as some files contain data descriptions at the top or end of the file which do not fit in the data's "table structure".

1.2. `.RData` files

`.RData` files are proprietary to `R` and are used to store `R` objects. A single `.RData` file is not limited to a single object, but can comprise many objects. To load a `.RData` file you simply use the function `load` and provide it with the file path. The function will load the object(s) directly into your workspace and you are good to go.

```
load("../data/gspc_ret.RData")
head(gspc.ret)

##                    [,1]
## 1950-01-04  0.011404562
## 1950-01-05  0.004747774
## 1950-01-06  0.002953337
## 1950-01-09  0.005889282
## 1950-01-10 -0.002927342
## 1950-01-11  0.003523135
```

1.3. Cleaning data

Many times, the data that you have loaded will not be "perfect" in the sense that there are missing values (`NA`) or some values that do not appear plausible. This subsection will

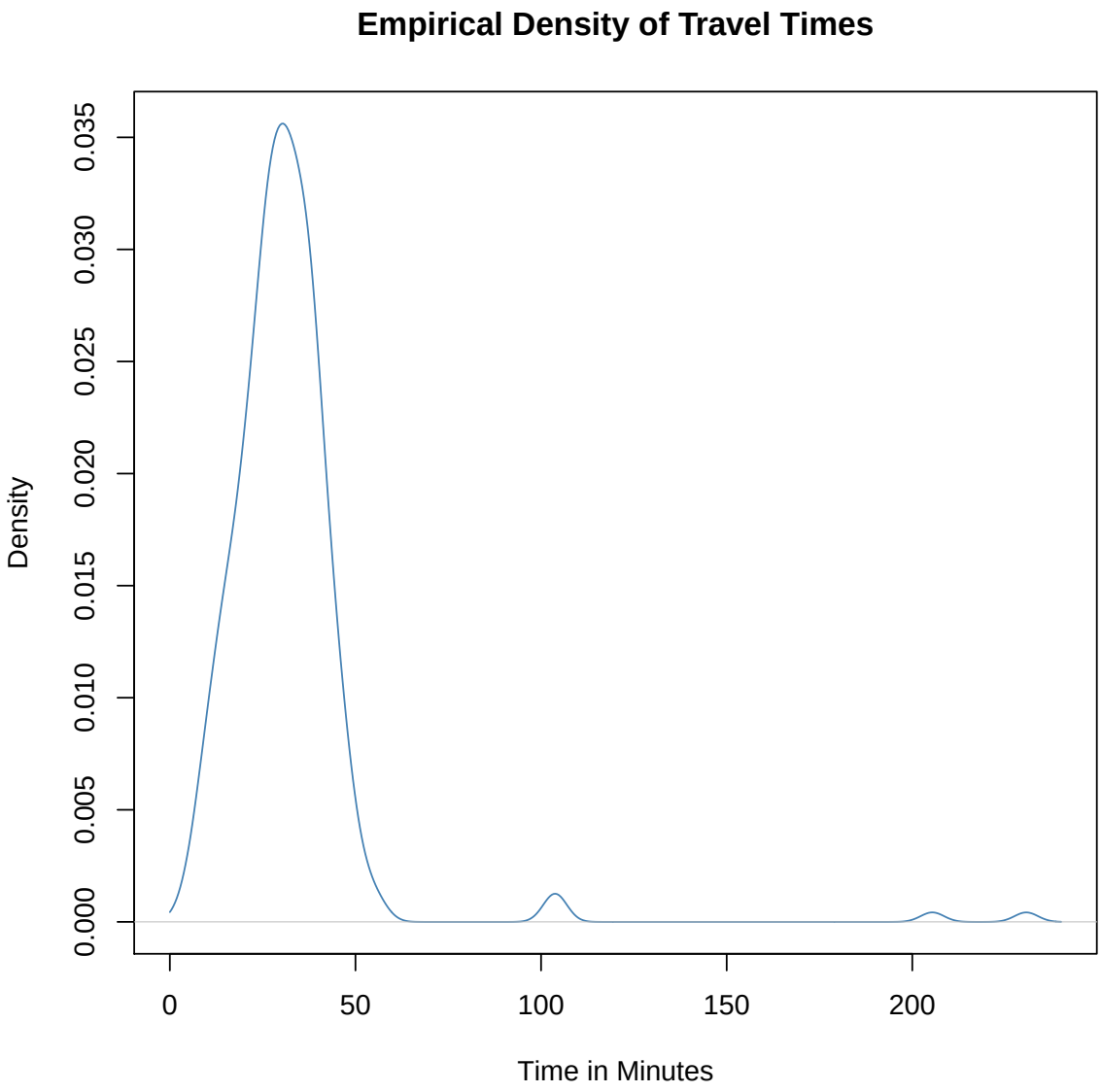
outline some approaches to handling such observations. Usually, there are two principle options available:

1. **Cleaning data by hand.** In some cases – and provided the dataset is not too big – implausible datapoints are still informative about the true value they likely should represent. For example, think of a dataset on the body height of people, where data is given in centimeters: 170, 164, 1.85, 192. Most probably, 1.85 does not mean 1.85 *centimeters*, but was accidentially entered in *meters* by the interviewee.

2. **Cleaning data through rules.** In some cases, cleaning data by hand is not practical, either because the dataset is simply too large, or because you have little intuition for how extreme reasonable values can be. In these cases, it usually is not sensible to consider each and every value by hand. Instead, it is common practice to identify outliers by considering the whole distribution of your data set, and then simply cut off the most extreme values based on some threshold. These thresholds are usually defined in terms of quantiles of the distribution – for example, one could define all values below the 1%-quantile and all values above the 99%-quantile of the data as too extreme to be reasonable. The most conservative option then would be to simply remove these values from the dataset. Alternatively, however, if we think that the *direction* of the suspicious value is still informative, we can also just set it to some lower, less extreme value: For example, you might encounter an outlier way above the usual level of other observations in the data set, but still think that the true value would have also been quite large and just got additionally inflated due to some measurement error. In this case, it might make sense to replace this excessively large value by the 99%-quantile of the dataset, with still constitutes a relatively high value, but not excessivly large (you could in principle pick any threshold you feel might be reasonable, the 99%-quantile is no magic number here).

To illustrate truncating outliers of a data set through quantiles, let's first create an ficticious data set of students' travel times from residences located in Vienna to the WU campus.

```r
# setting a seed makes random draws from a distribution replicable
set.seed(213)

# simulated data set of travel time in minutes
travel_times <- rnorm(n=295, mean=30, sd=10)
# add 5 larger-than-usual values
travel_times <- c(travel_times, 103.2, 104, 104.1, 205.4, 230.7)
plot(density(travel_times, from=0), col="steelblue",
     main="Empirical Density of Travel Times",
     xlab="Time in Minutes")
```



We now could use any quantile of our empirical distribution of travel times and consider all values above as outliers. These outliers then can either be removed from our data, or set to the threshold value.

```
q99 <- quantile(travel_times, probs=0.99) # 99% quantile

# throwing away extreme values
travel_times1 <- travel_times[travel_times<=q99]

# replacing extreme values by threshold
travel_times2 <- travel_times
travel_times2[travel_times2>q99] <- q99
```

To illustrate the handling of missing values, consider the following example:

```
my_data <- data.frame(a=c(1, NA,2), b=4:6, c=10:12)
my_data

##    a b  c
## 1  1 4 10
## 2 NA 5 11
## 3  2 6 12
```

We can use the function `is.na()` to detect missing values.

```
is.na(my_data)

##          a     b     c
## [1,] FALSE FALSE FALSE
## [2,]  TRUE FALSE FALSE
## [3,] FALSE FALSE FALSE
```

To remove missing values right away, use `na.omit()`. However, have a look how this function works on a single vector and on a `data.frame` like `my_data`:

```
vector_a <- my_data$a
vector_a

## [1]  1 NA  2

na.omit(vector_a)

## [1] 1 2
## attr(,"na.action")
## [1] 2
## attr(,"class")
## [1] "omit"

# the function has returned some additional infos that can be ignored
# it works as expected and has only left 1 and 2, but removed NA

na.omit(my_data)
```

```
##   a b  c
## 1 1 4 10
## 3 2 6 12

# na.omit removes the whole row
# if it finds a single NA in a particular row!
```

Not that for data in the form of *time series*, it can be sensible to use information from the previous and/or the next value to deal with a missing value in the time series – for details, see the functions `na.locf()` and `na.approx()` in section 3.3.3.

2. Basic cross-sectional data visualization

In the following a few selected methods for cross-sectional data visualization are presented. We encounter cross-sectional data when we look at different entities at a single point in time. For example, the market value of all S&P 500 companies on the 1st of January. Let us take a brief look at the data structure for this section before we take a look at different methods of visualization.

```
head(cs.data)
```

```
##                            conm          ret    prc          ME       BE
## 1       J & J SNACK FOODS CORP  0.079791039 152.47  2850.73161  745.027
## 2                  DGSE COS INC -0.111111075   0.72    19.38528    7.756
## 3                   PLEXUS CORP  0.023903687  59.54  1926.77397 1025.939
## 4 ROCKY MOUNTAIN CHOC FACT INC  0.036771297  11.44    67.55320   18.829
## 5        PRICE (T. ROWE) GROUP -0.038132120 116.09 28230.76531 5824.400
## 6                  AMERESCO INC  0.004184117  12.00   354.79200  337.204
##   Industry    ID
## 1        4 10026
## 2        6 10028
## 3        4 10032
## 4        4 10044
## 5        8 10138
## 6       10 10158
```

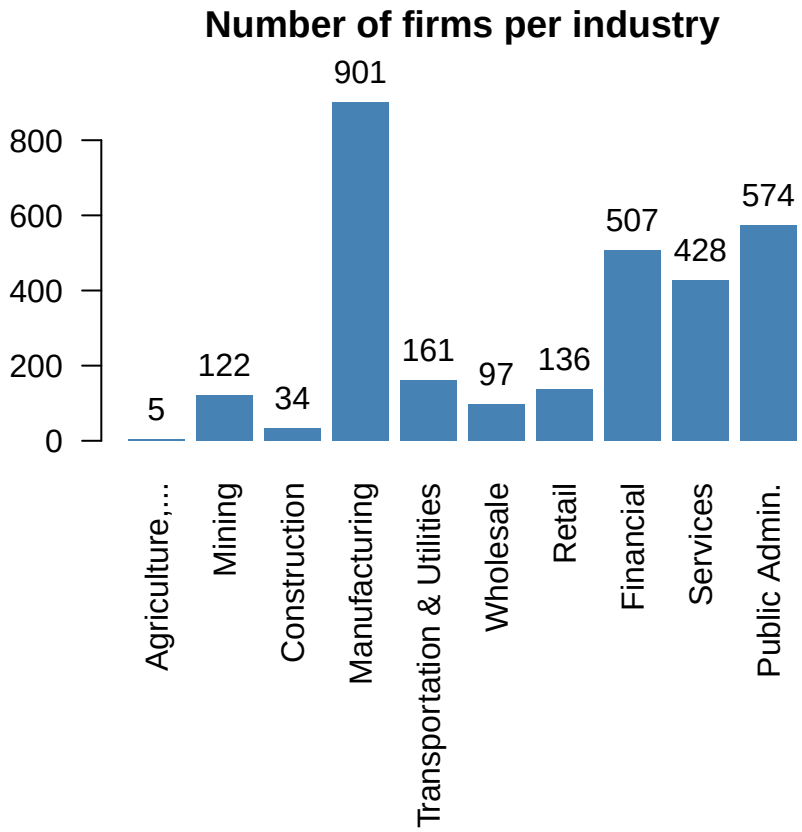
The data set contains the market value (ME) and book value (BE) of equity, the share price and last month's return as of June 2018 as well as an ID, the company name and an industry classifier. We will have a look at bar plots, histograms, box plots, scatter plots and QQ-plots as a special form of scatter plots. You should already know the use cases for all of these plots from your previous classes.

2.1. Bar plots

One of the most common ways of visualizing cross-sectional data is the bar plot. Usually bar plots are used to visualize counts of different groups. For example, an analyst could be interest in the distribution of companies according to industry.

```
##Barplot
##Let's plot the number of firms per industry
par(mar=c(12,4.1,4.1,2.1))
##Adjust plot-margin in order to display the x-axis labels properly
par(xpd=TRUE)
##Allow the graphics device to draw outside of the plot area
data.freqs<-table(cs.data$Industry) #Get the number of firms per industry
##Barplot: use las=2 to write the x-axis vertically instead of horizontally
bp<-barplot(data.freqs, col = "steelblue", border = FALSE, las=2,
            main="Number of firms per industry",
            names.arg = c("Agriculture,...","Mining",
                          "Construction","Manufacturing",
```

```
                          "Transportation & Utilities",
                          "Wholesale","Retail","Financial",
                          "Services","Public Admin."))
text(x = bp,y = data.freqs,labels = data.freqs, pos=3)
##Write the number of firms in each industry above the respective bars
##Reset graphcis device
par(xpd=FALSE)
par(mar=c(5.1,4.1,4.1,2.1))
```



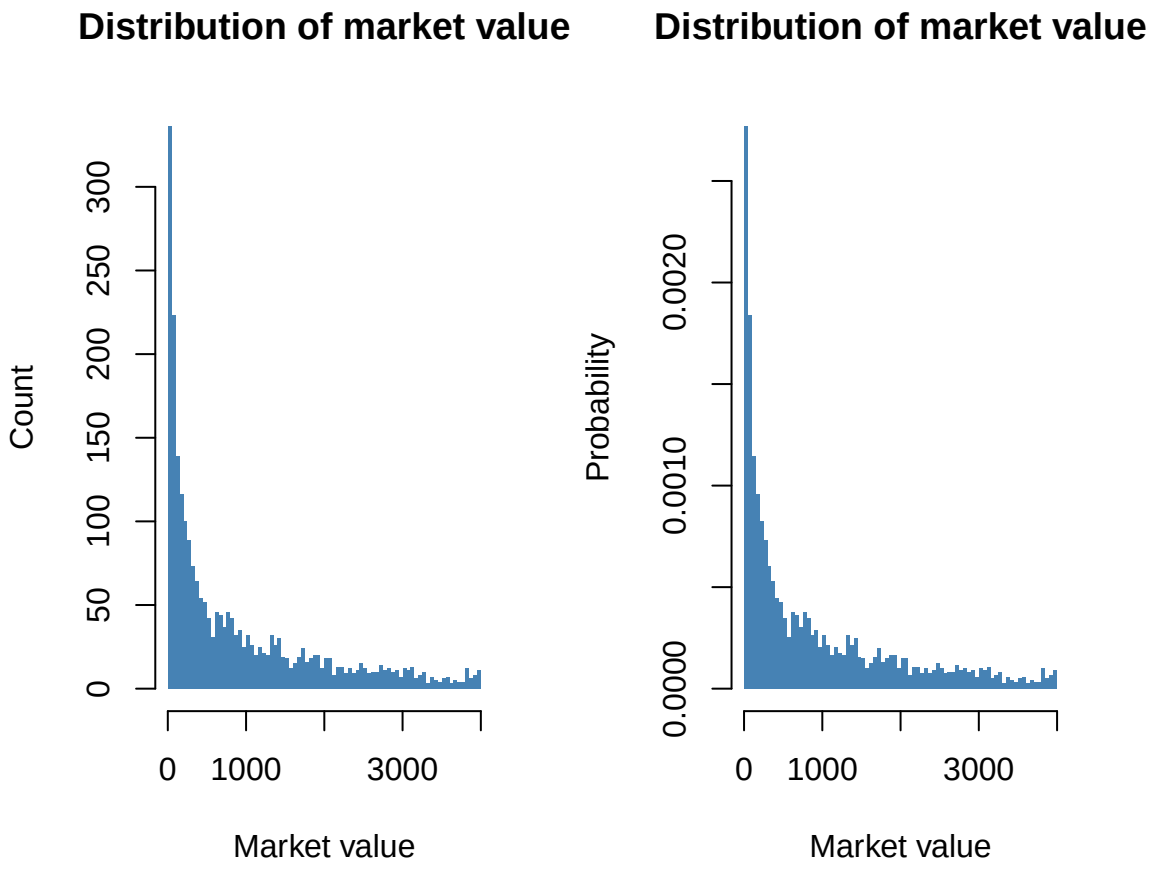
The bar plot should be used if the data dimension represented on the x-axis is either nominal or ordinal. If the data is ordinal, then one usually plots the data increasing from left to right.

2.2. Histogram

The histogram looks very similar to a bar plot but is different in a very important way. Namely, the x-axis is continuous and thus we can at least approximately visualize the distribution of a continuous sample of data. By sorting data into bins we can represent each bin by a bar and the height of the bar indicates the absolute or relative amount of data in that bin. If we increase the number of bins the approximation of the distribution becomes better. In `R` the argument `breaks` specifies the number of bins used. The `breaks`

argument is actually more flexible, see `?hist` for details. For `freq=TRUE` the height of a single bar indicates the absolute number of data points in the corresponding bin. For `freq=FALSE` the heights of the bars a normalized in a way that the histogram has a total area of one representing a density. For example, we can plot the distribution of market value of equity of all small firms in June 2018.

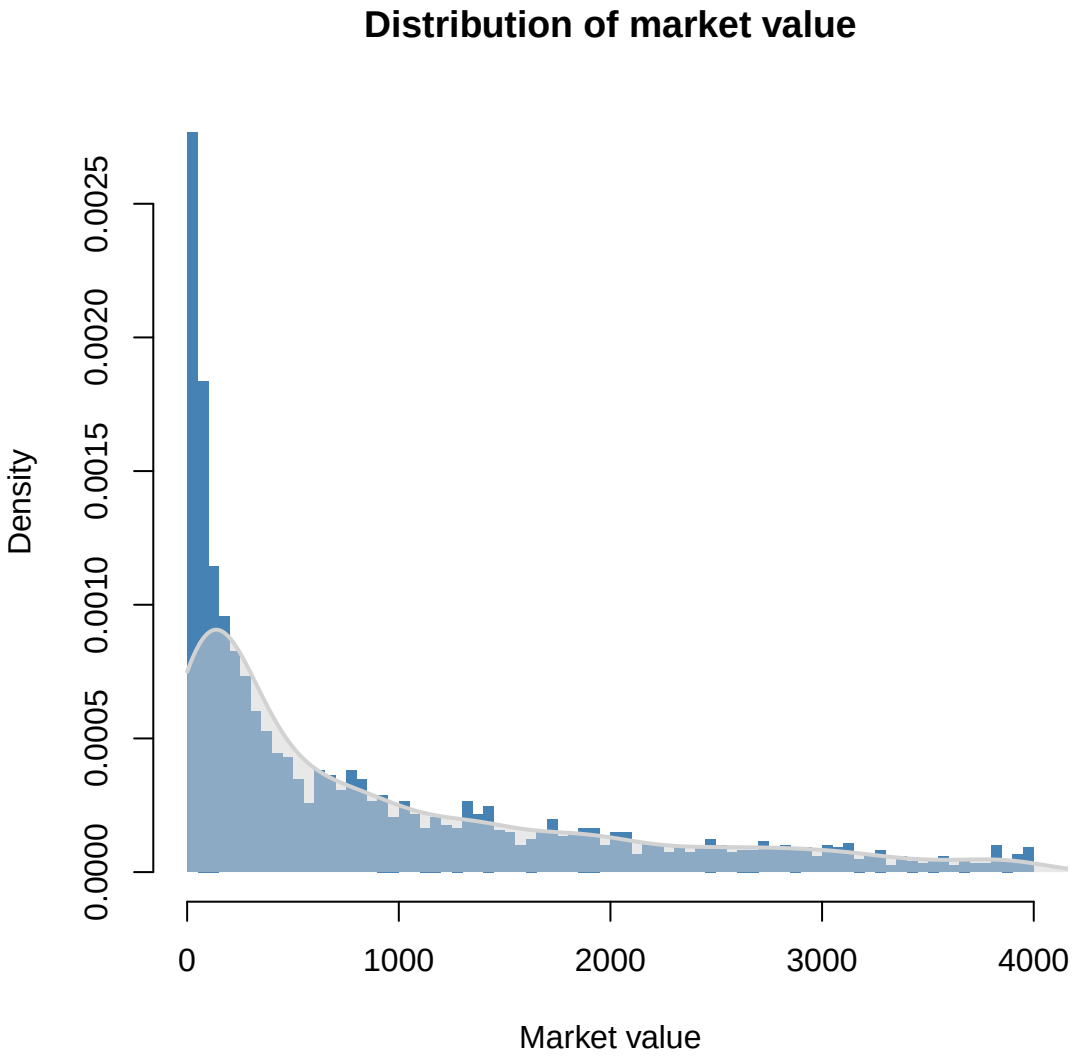
```
##Histogram
##Let's plot the distribution of market values for small firms
##using a histogram

me<-cs.data$ME[cs.data$ME<4000]
par(mfrow=c(1,2)) # draw 2 plots next to each other
# [par(mfrow=c(2,1)) would plot one histogram above the other]
hist(me, breaks = 100, freq = TRUE,
     main="Distribution of market value",xlab="Market value",
     border=FALSE, col="steelblue", ylab="Count")
hist(me, breaks = 100, freq = FALSE,
     main="Distribution of market value",xlab="Market value",
     border=FALSE, col="steelblue", ylab="Probability")
par(mfrow=c(1,1))
```



To make things look even more pretty we can estimate the density and overlay it.

Here, the function `density()` estimates the empirical density of our data, the argument `from=0` tells the function that our data is restricted to non-negative values only (i.e. to start the density estimation from 0).

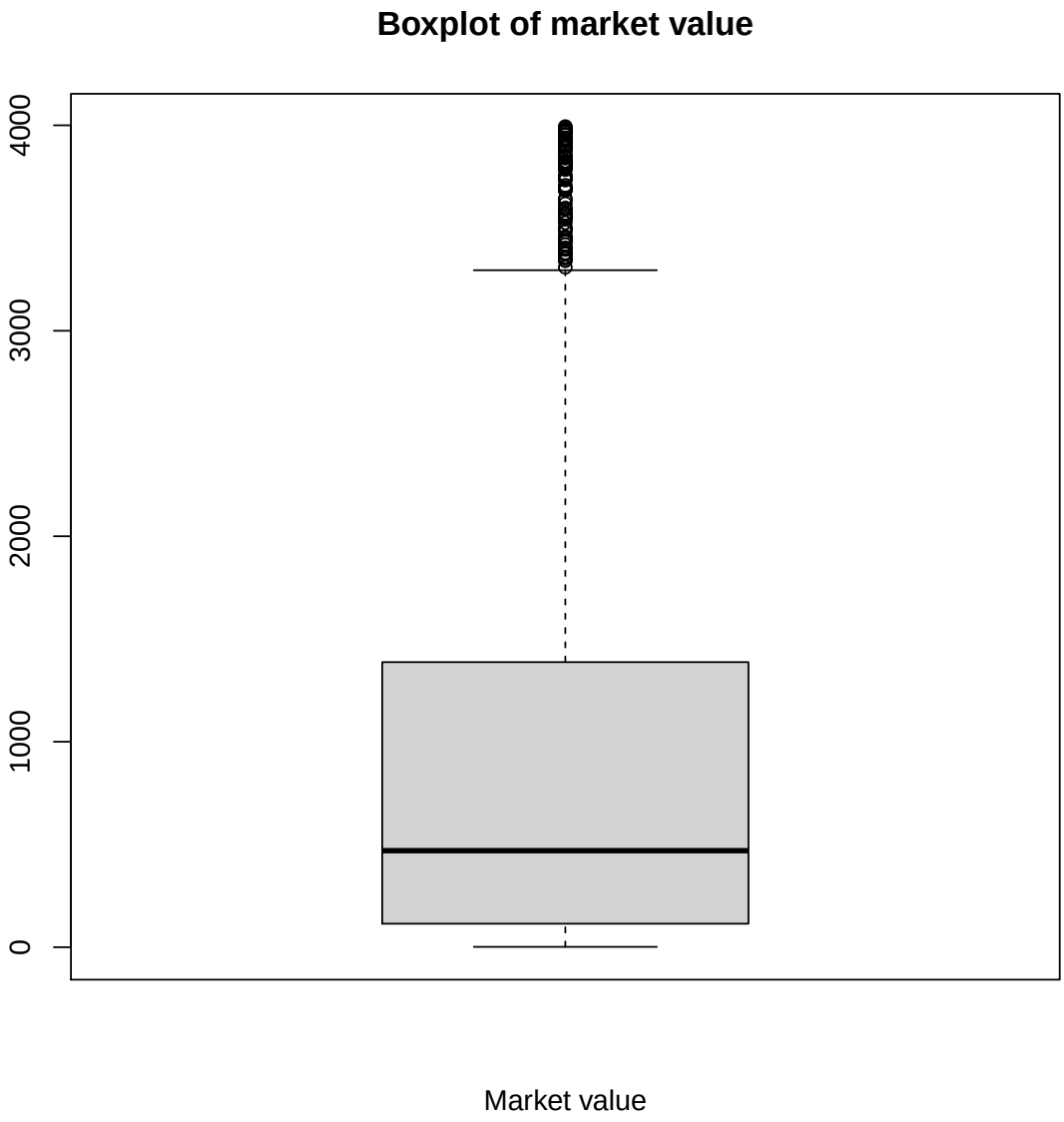
```
##Histogram
##Let's plot the distribution of market values for small firms
##with a histogram and add a density line
me<-cs.data$ME[cs.data$ME<4000]
den<-density(me,from = 0)
# extract x and y coordinates from estimated density
# these coordinates are then used to draw the semi-transparent polygon
cord.x<-c(den$x[1],den$x,rev(den$x)[1])
cord.y<-c(0,den$y,0)

hist(me, breaks = 100, freq = FALSE,
     main="Distribution of market value",xlab="Market value",
     border=FALSE, col="steelblue")
lines(den, lwd=2,col="lightgrey")
polygon(x = cord.x, y = cord.y,
        col=adjustcolor("lightgrey",alpha.f = .5), border = FALSE)
```



2.3. Box plot

Another way of looking at distributional properties of the data is the box plot. As with most plotting methods R has a ready to use function for that. For example, let us take a look at the same data as before, i.e. the market value of equity of small firms.

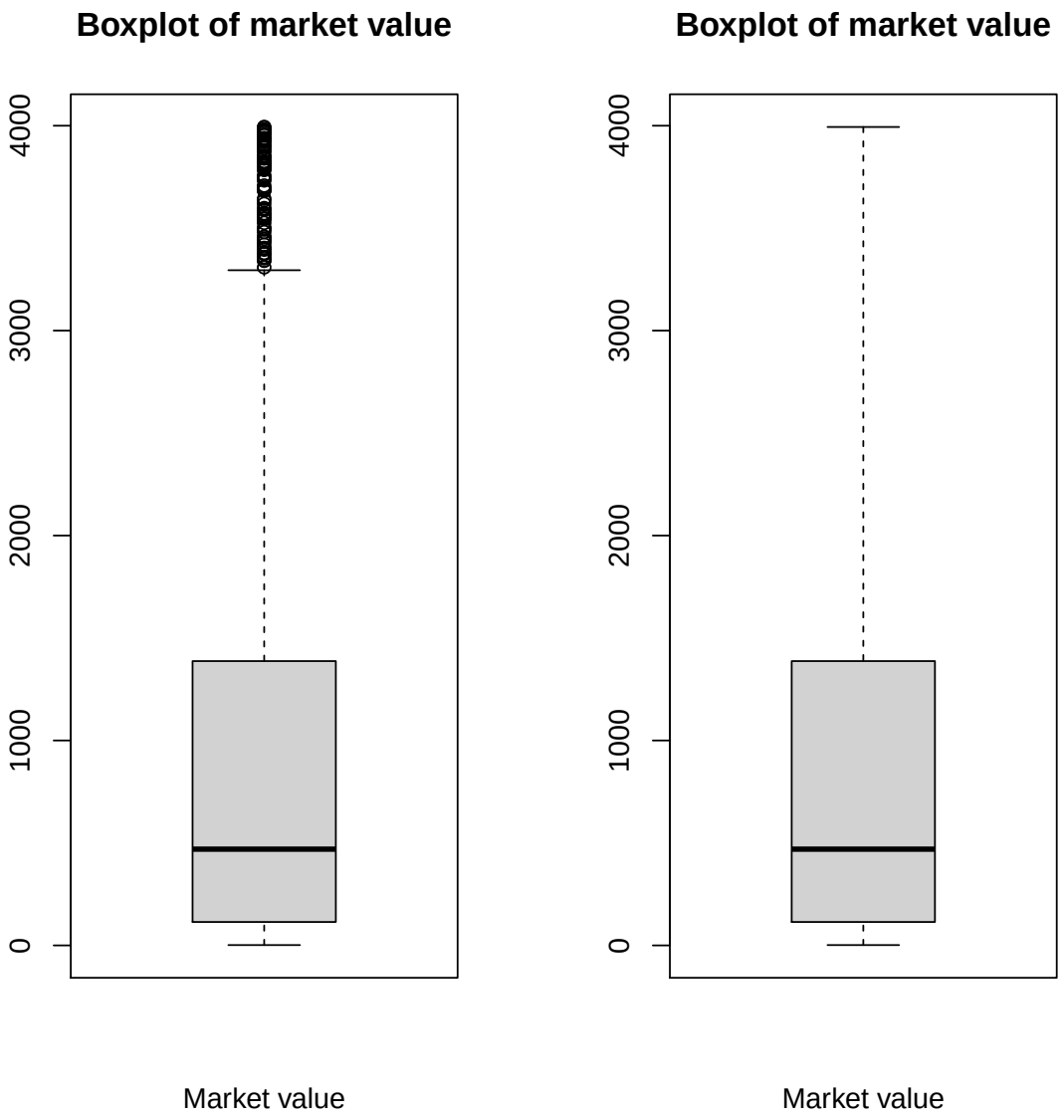
```r
##Boxplot
me<-cs.data$ME[cs.data$ME<4000]
boxplot(me, main="Boxplot of market value",xlab="Market value")
```



The box plot or box and whiskers plot visualizes the data through quartiles. The bottom of the box indicates the lower quartile, the line through the box the 50% quantile, i.e. the median, and the top of the box indicates the upper quartile. The whiskers extending from the box give us an idea about the variability in the lowest and highest quartiles. Dots indicate outliers. If we set `range=0`, then the whisker cover the whole sample.

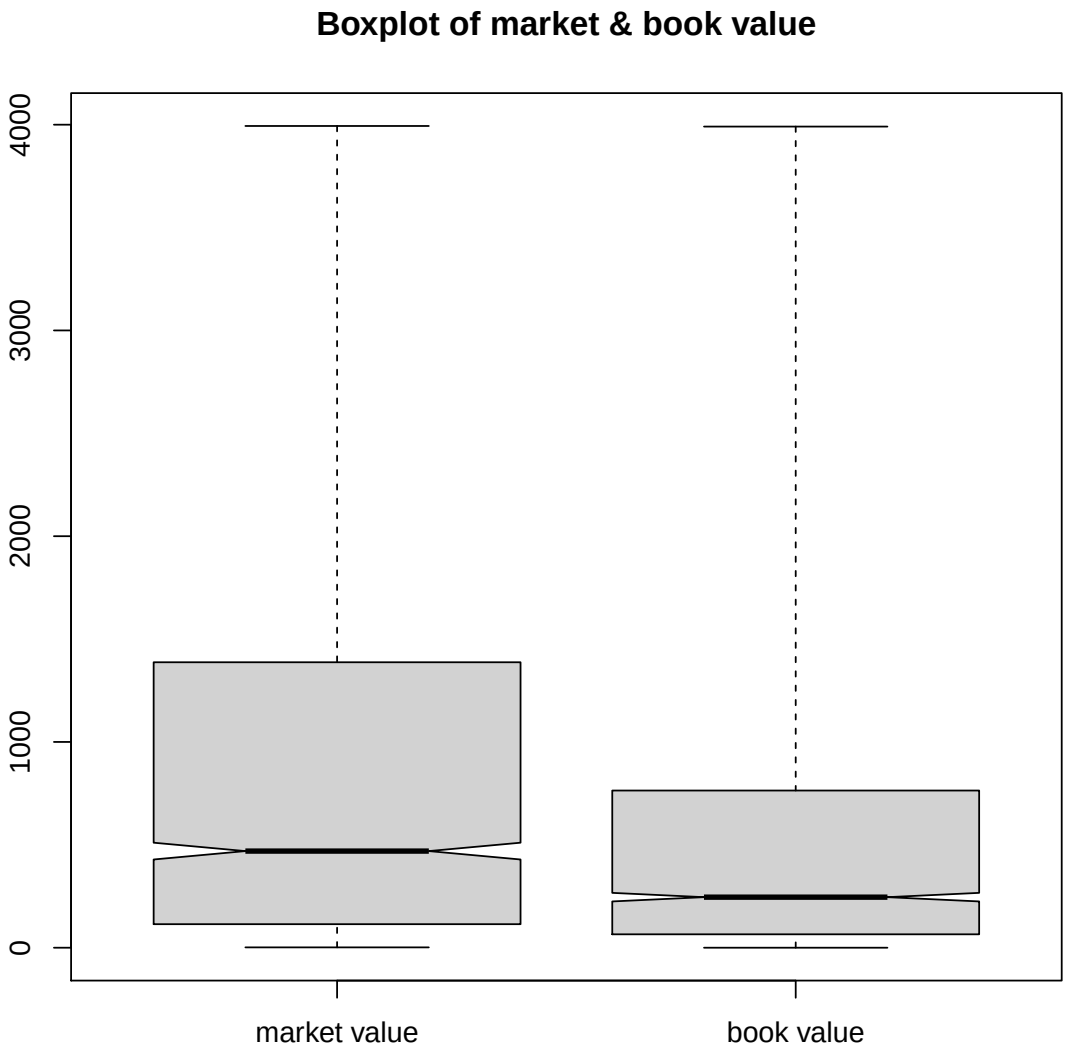
```r
##Boxplot
par(mfrow=c(1,2))
me<-cs.data$ME[cs.data$ME<4000]
boxplot(me, main="Boxplot of market value",xlab="Market value")
boxplot(me, main="Boxplot of market value",xlab="Market value", range = 0)
```

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



We can also plot multiple variables next to each other, e.g. market value and book value of equity. Moreover, by using `notch=TRUE`, we can get an idea of how different the medians of both variables are. If the notches do not overlap this is a strong indication for a difference in medians.

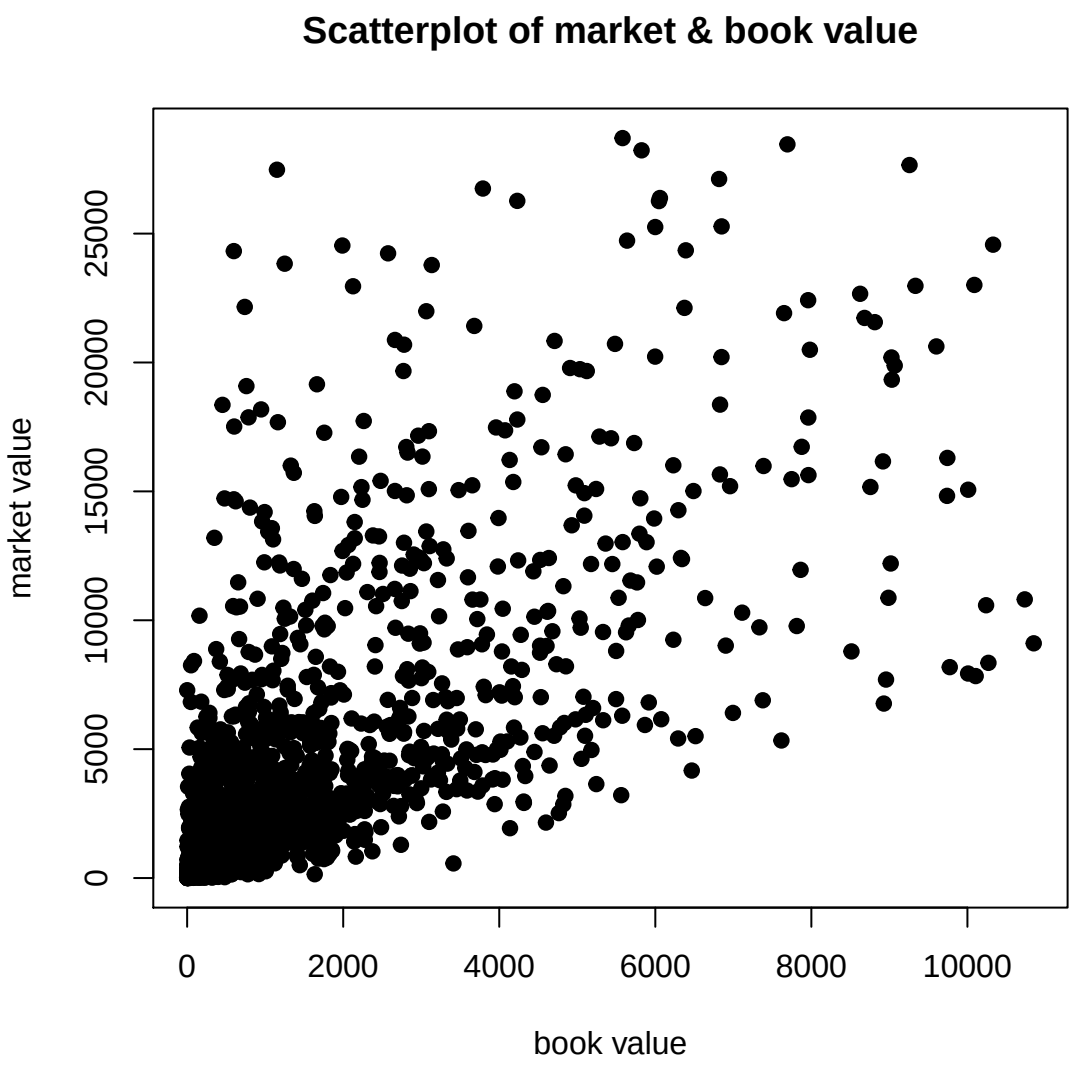
```
##Boxplot
me<-cs.data$ME[cs.data$ME<4000]
be<-cs.data$BE[cs.data$BE<4000]
boxplot(me, be, main="Boxplot of market & book value",
        names=c("market value", "book value"), notch = TRUE, range = 0)
```



2.4. Scatter plot

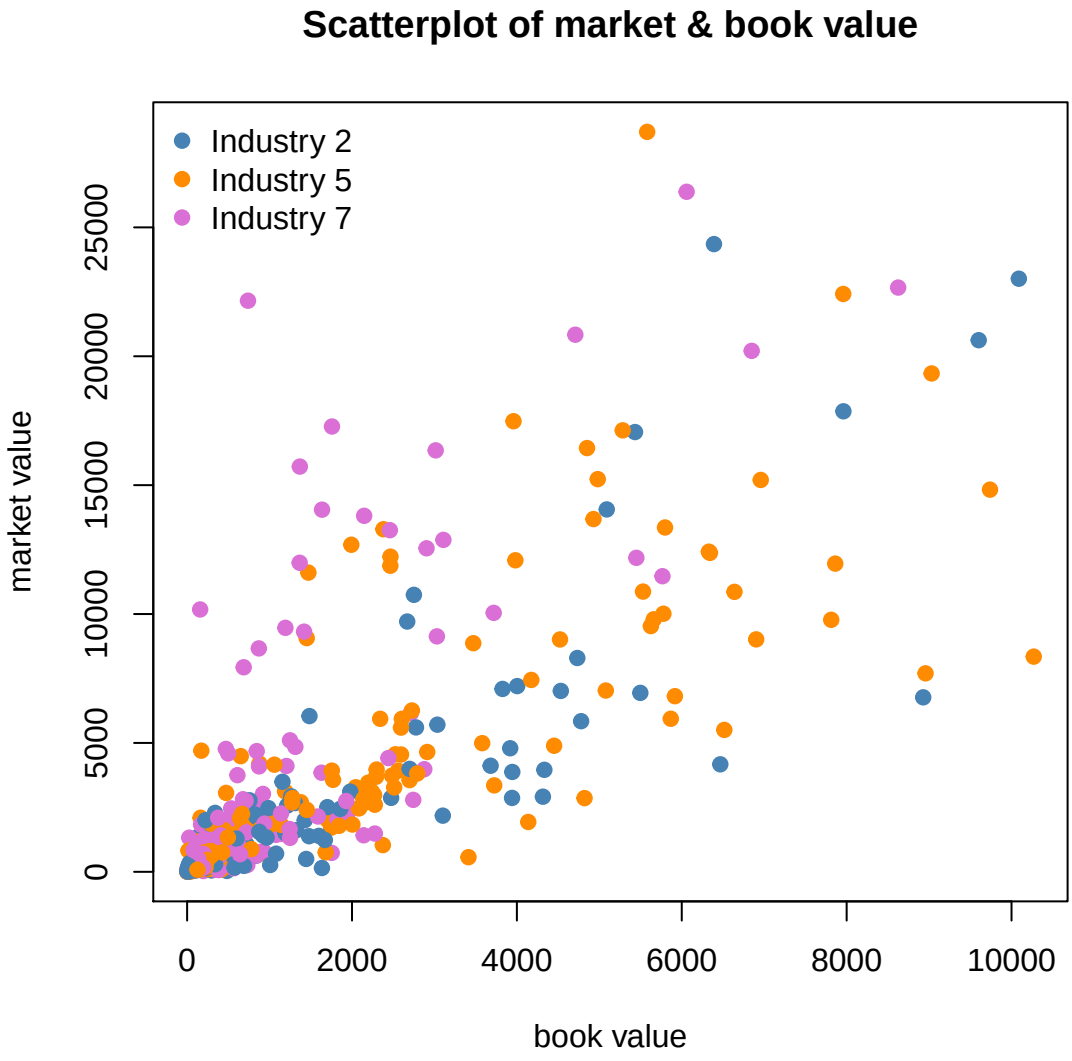
Sometimes we are interested in the relationship between two variables. Scatter plots help us to understand that relationship by representing one variable on the x- and one on the y-axis. If we think about book and market value of equity of a firm we would suspect that the relationship is a positive one (in a correlation sense). Thus we would expect firms with a high book value of equity to have a high market value of equity and vice versa. To visually check our claim we can utilize a scatter plot. Each point in the scatter plot represents a firm in our sample and is placed in such a way that the x-axis value is equal to the book value of equity and the y-axis value equals the market value of equity of that firm.

```
##Boxplot
me<-cs.data$ME
be<-cs.data$BE
plot(y=me, x=be, main="Scatterplot of market & book value",
        ylab="market value", xlab="book value", pch=19)
```



We can bring additional information to this plot by color coding firms according to their industry. To make things look neat, we only plot firms belonging to industries 2, 5 and 7.

```r
##Boxplot
me<-cs.data$ME[cs.data$Industry%in%c(2,5,7)]
be<-cs.data$BE[cs.data$Industry%in%c(2,5,7)]
ind<-cs.data$Industry[cs.data$Industry%in%c(2,5,7)]
indcols<-rep(NA_character_, length(ind))
indcols[ind==2]<-"steelblue"
indcols[ind==5]<-"darkorange"
indcols[ind==7]<-"orchid"
plot(y=me, x=be, main="Scatterplot of market & book value",
        ylab="market value", xlab="book value", pch=19, col=indcols)
legend("topleft", legend = c("Industry 2","Industry 5","Industry 7"),
       col=c("steelblue", "darkorange", "orchid"), pch=19, bty="n")
```

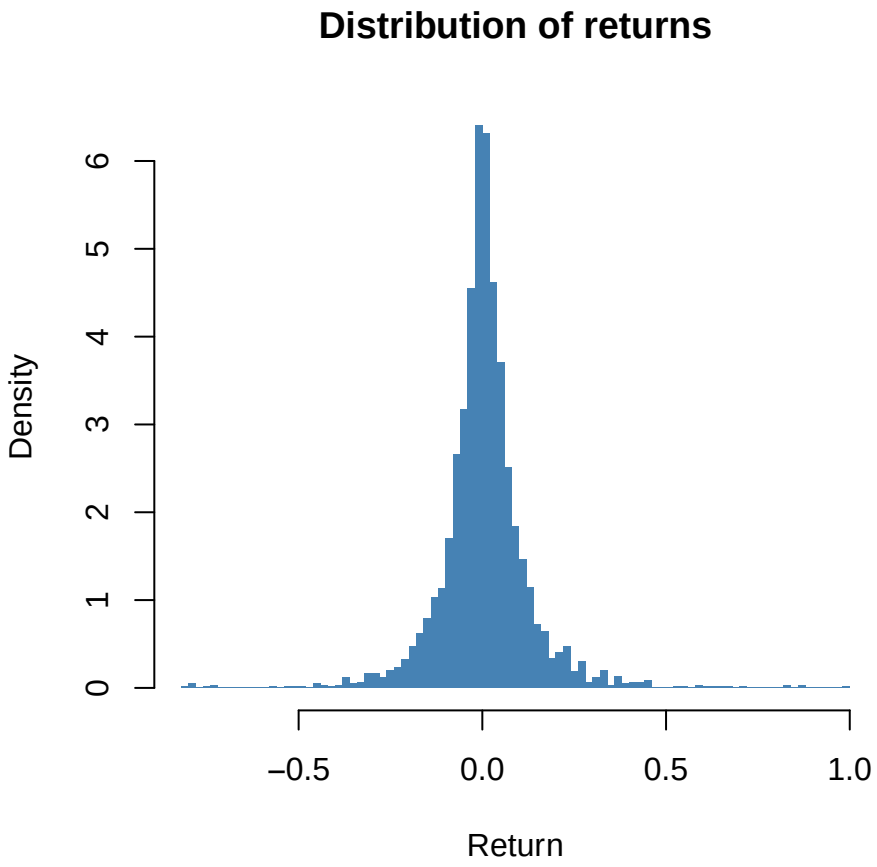


2.5. QQ-plot

A special kind of scatter plot is the quantile-quantile plot (QQ-plot). We can use it to visually asses how well the data fit a theoretical distribution. On the x-axis the theoretical

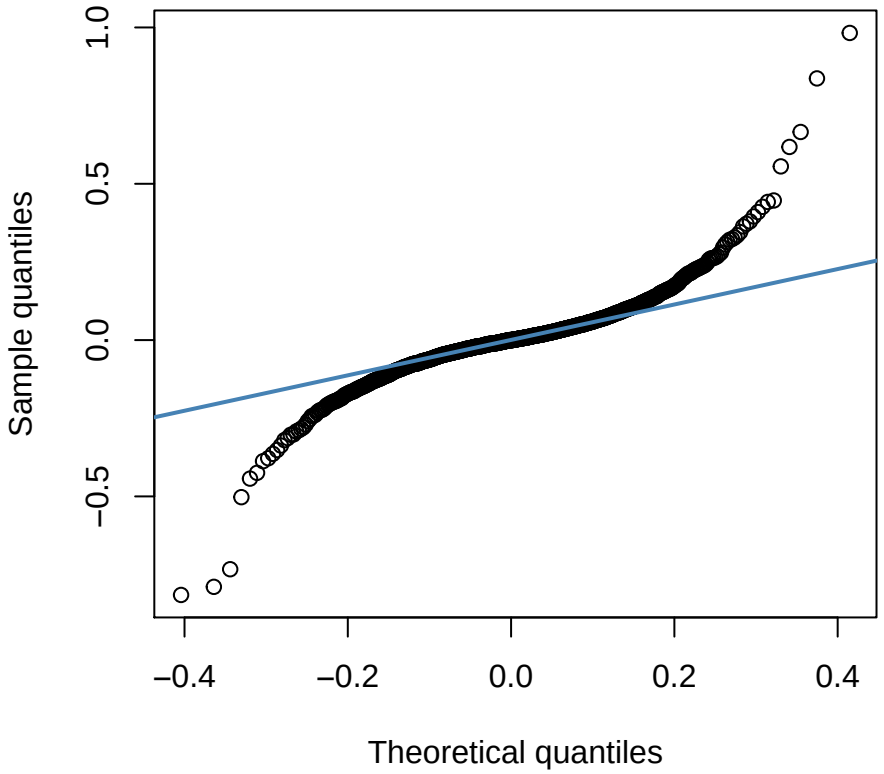
quantiles are plotted and on the y-axis the sample quantiles are plotted. For example, consider the following problem: you look at the histogram of last month's returns (see below) and wonder what distribution this resembles. Seeing the symmetric shape you immediately think of a bell and of Gauss. This might look normally distributed to you at the first glance and you want to use a QQ-plot to check your guess.

```
##Histogram
ret<-cs.data$ret
hist(ret, breaks = 100, freq = FALSE,
     main="Distribution of returns",xlab="Return",
     border=FALSE, col="steelblue")
```



The function `qqplot` has two main inputs. Remember on the x-axis we want the theoretical quantiles and on the y-axis the sample quantiles. To get the theoretical quantiles of a normal distribution you can use the function `qnorm`. But before plotting sample and theoretical quantiles against each other we first need to specify the normal distribution, i.e. we need to choose the location and scale parameters — the mean and the standard deviation. To compute the theoretical quantiles we thus use the sample mean and the sample standard deviation as parameters for our theoretical quantiles. Now, if the data is indeed normally distributed with mean and standard deviation as specified above, then all quantiles should coincide and the points should form a line from the bottom left to the top right. We can plot this line with the `qqline` function. The closer the points are to this line the better the fit.

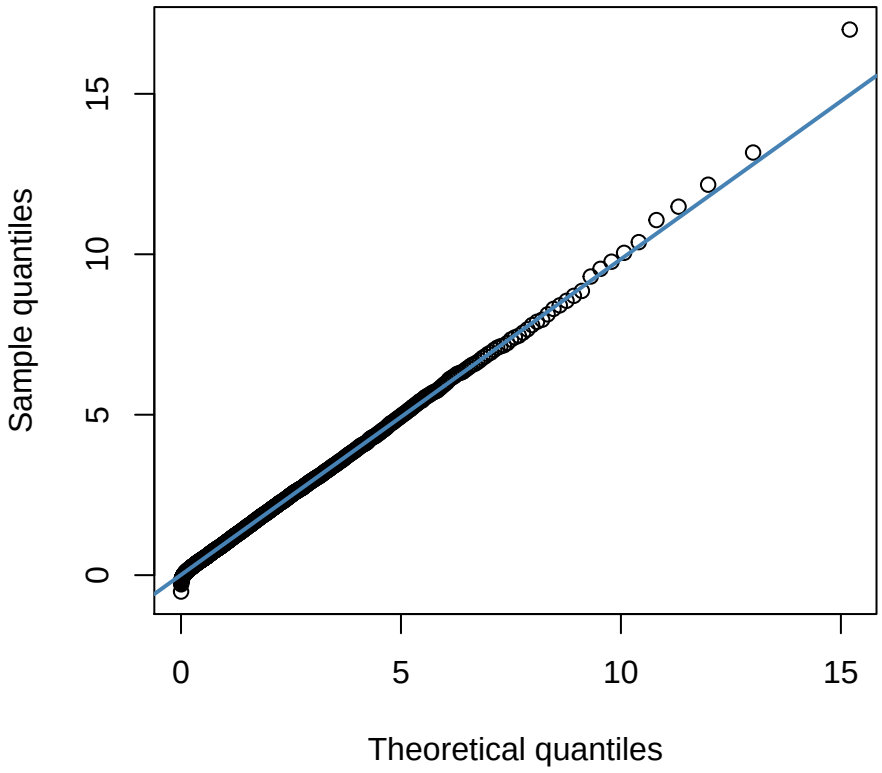
```
##Histogram
ret<-cs.data$ret
theoretical_quantiles<-qnorm(p = ppoints(1000), mean = mean(ret),
                             sd = sd(ret))
qqplot(x = theoretical_quantiles, y = ret,
       ylab="Sample quantiles", xlab="Theoretical quantiles")
qqline(y = ret, distribution = function(p){ qnorm(p,mean(ret), sd(ret))},
       col="steelblue", lwd=2)
```



In the above figure we see that the points fit the line in the center of the distribution (in the range from -0.1 to 0.1) rather well but not in the tails — a phenomenon you will observe very often. So, we can conclude that our data are most likely not drawn from a normal distribution.

To see how a QQ-plot looks like with data fitting a theoretical distribution we first draw from this distribution, add a little bit of noise and then draw a QQ-plot.

```r
##Histogram
set.seed(15)
draws<-rgamma(10000, shape = 1, scale = 2)
# drawing from a Gamma-distribution
draws<-draws+rnorm(10000, mean = 0, sd = 0.15) #adding a bit of noise
theoretical_quantiles<-qgamma(p = ppoints(1000), shape = 1, scale = 2)
qqplot(x = theoretical_quantiles, y = draws,
       ylab="Sample quantiles", xlab="Theoretical quantiles")
qqline(y = draws, distribution = function(p){
  qgamma(p, shape = 1, scale = 2)},
       col="steelblue", lwd=2)
```



We see that the points fit the line almost perfectly. This is a strong indication that the data are indeed distributed according to theoretical distribution, but be aware that this is not a formal test in a statistical sense!

3. The xts package

In this section, you will learn how to work with the **xts** package, a super useful tool to handle timeseries in R. The package employs its own object class, which – unsurpirsingly – is called **xts**.

To get started, let's first load the package and some data to use it on. In this section, we will use daily data (trading days) on German government bond yields at maturities of 2Y, 5Y, 10Y and 30Y.

```
library(xts)
head(d)

##         Date     X2Y     X5Y    X10Y    X30Y
## 1 11.12.2000 4.65575 4.67264 4.85954 5.58972
## 2 12.12.2000 4.65641 4.67391 4.88013 5.59810
## 3 13.12.2000 4.59706 4.60987 4.83391 5.55546
## 4 14.12.2000 4.52651 4.57871 4.81444 5.48557
## 5 15.12.2000 4.50157 4.52840 4.77875 5.45185
## 6 18.12.2000 4.47767 4.48716 4.75643 5.46294
```

As you can see, the data (**d**) is a **data.frame** object that contains the dates (object of class **factor** or **character**) in its first column and the data on the yields at the respective maturities in the other columns (object of class **numeric**).

3.1. Converting to an xts object

To convert this to an **xts** object, we need to tell R that the first column of **d** contains the dates, and the other four columns the data to which the dates should be assigned to. To do so, we first need to convert the column with the dates into a format that R can understand as a date: This is done by the **as.Date()** function. In particular, you need to specify the date-**format** that the dates in the first column of **d** have. This is achieved by writhing a **%d**, **%m** and **%Y** respectively where the numbers for the day, month and year appear in a character string in the first colum. If the date is given as 11.08.1994 (Austrian format), for example, you would tell **as.Date** that the **format** is **"%d.%m.%Y"**. If the date is given as 1994/08/11, you would specify this as **"%Y/%m/%d"**.

Once the dates are in an approporate format, **d** can easily be converted to an **xts** object by using the function **xts()** that takes the **numeric** data on yields and the dates in the appropriate **Date** format as arguments:

```
#converting first column with dates into suitable format
t_index <- as.Date(d[,1], format="%d.%m.%Y")
class(t_index)

## [1] "Date"

d <- xts(d[,-1], order.by=t_index)  #dropping first column of data
                                    #only containing dates
                                    #(in wrong format)
```

Let's check the object class of `d`:

```
class(d)

## [1] "xts" "zoo"
```

It is indeed `xts` (as well as `zoo`)! Impressive, right?

```
head(d)

##                 X2Y     X5Y    X10Y    X30Y
## 2000-12-11 4.65575 4.67264 4.85954 5.58972
## 2000-12-12 4.65641 4.67391 4.88013 5.59810
## 2000-12-13 4.59706 4.60987 4.83391 5.55546
## 2000-12-14 4.52651 4.57871 4.81444 5.48557
## 2000-12-15 4.50157 4.52840 4.77875 5.45185
## 2000-12-18 4.47767 4.48716 4.75643 5.46294
```

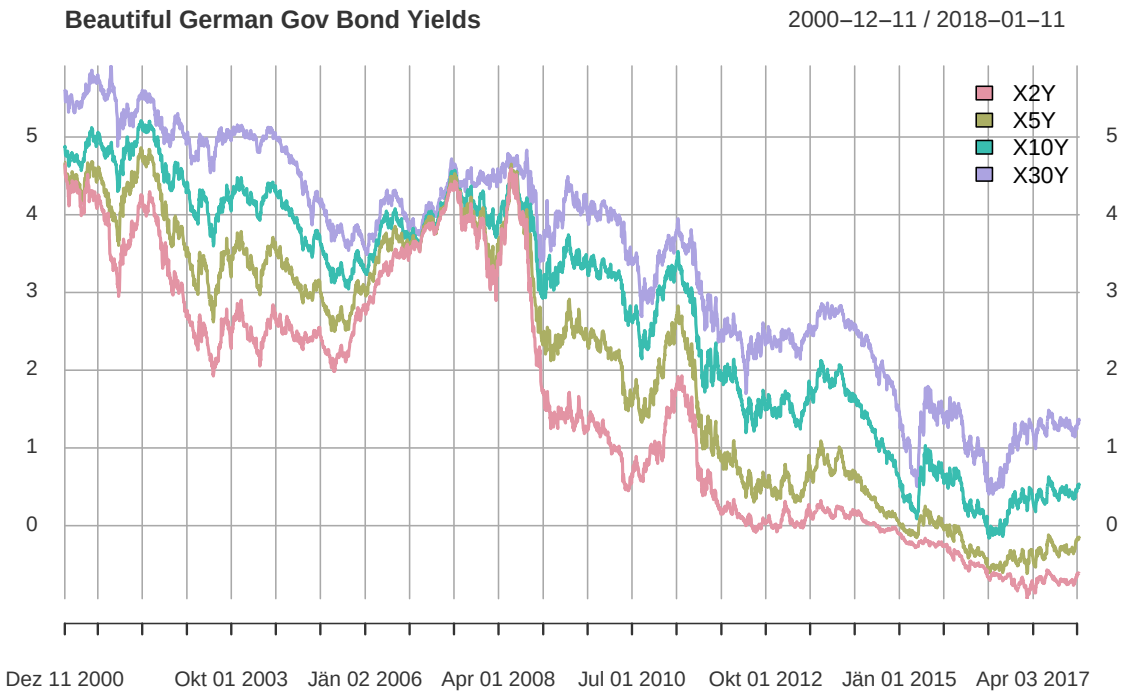
You can quickly count the number of years/months/weeks covered by an `xts` timeseries by using `nyears()`/`nmonths()`/`nweeks()`:

```
nyears(d)

## [1] 19

nmonths(d)

## [1] 206

nweeks(d)

## [1] 892
```

3.2. Plotting with `xts`

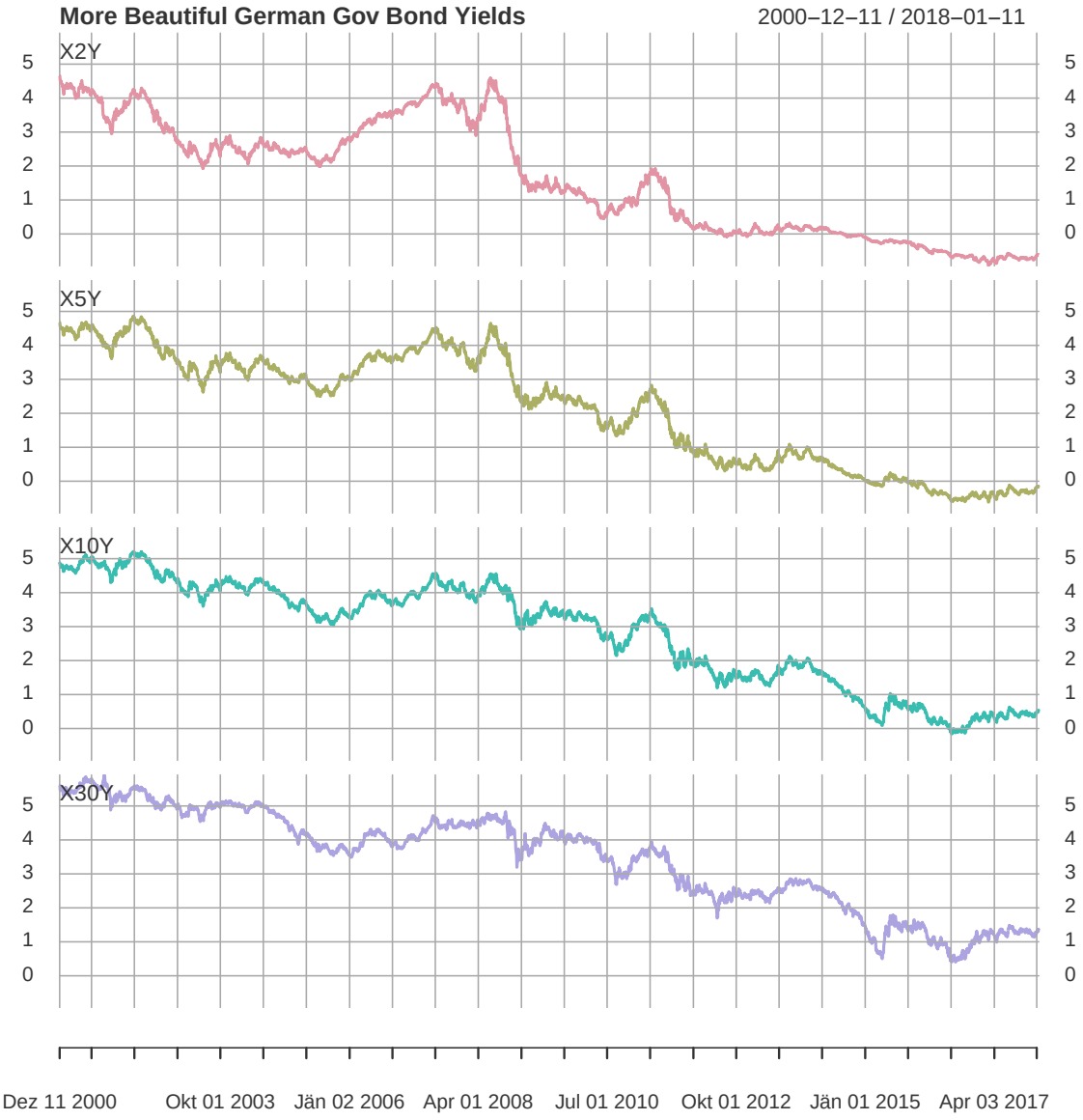
With `d` as an `xts` object, we can now do super cool stuff really easily. Like plotting all four timeseries at once:

```
library(colorspace) #package to get access to nicer colors,

    ## Warning:  package 'colorspace' was built under R version 4.0.3

                    # you could as well stick with the default
plot(d, main="Beautiful German Gov Bond Yields",
     col=rainbow_hcl(4),#gives you 4 colors from HCL colorspace
     legend.loc="topright") #creates auto-legend
```



Or plotting them in four separate panels:

```
plot(d, main="More Beautiful German Gov Bond Yields",
     col=rainbow_hcl(4),
     multi.panel=4)
```

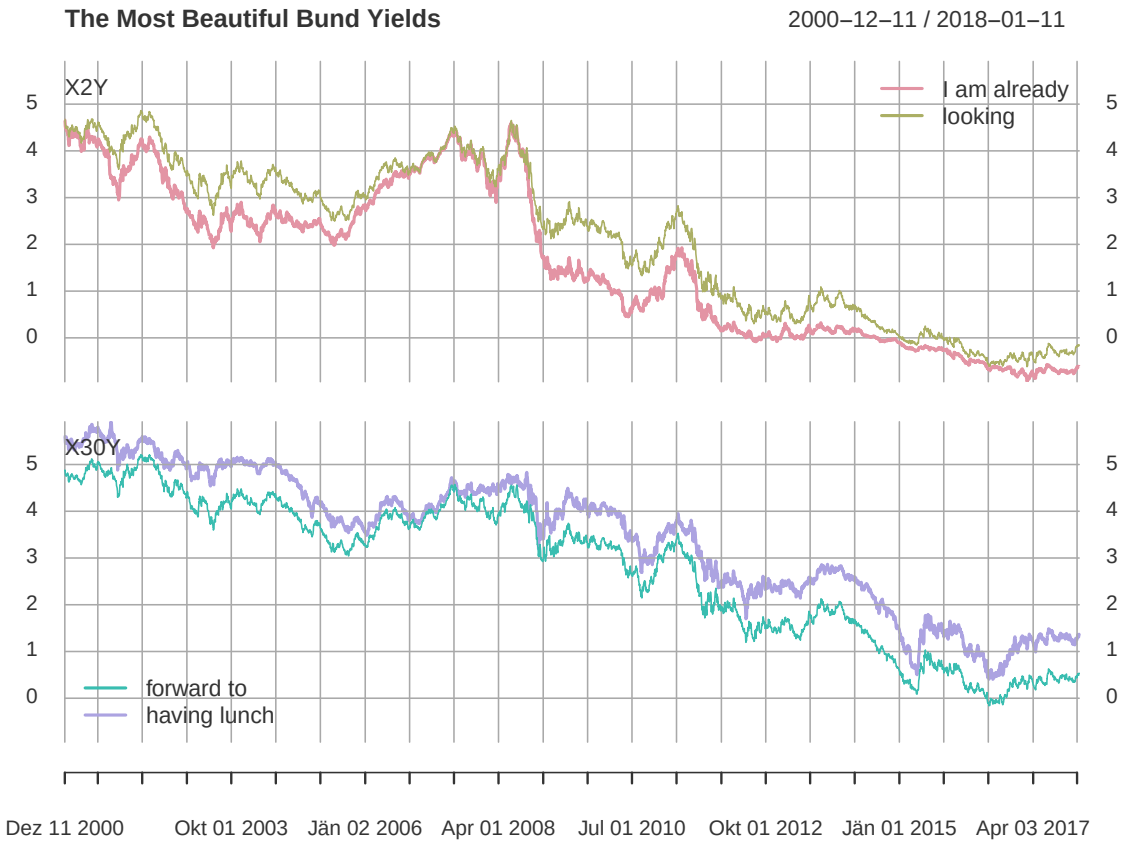


Note: By default, all panels drawn using `multi.panel=TRUE` will share the same scaling of the y-axis. If individual timeseries vary strongly in sclaing (magnitude), you might prefer setting the additional argument `yaxis.same=FALSE` – this way, the y-axis scalings will adjust to the individual timeseries drawn in the different panels.

Or plotting them in just *two* separate panels. This is achieved by plotting only two timeseries in a different panel, and then adding the remaining two timeseries to the panels using the `lines()` function. You can easily add `xts` timeseries to an existing `xts` plot by using `lines(youradditionalxtstimeseries)`. If you have a mulipanel `xts` plot, you also need to specify on which panel you want to draw the additional timeseries using the argument `on`. In addition, you can also add a legend more sophisticated than the auto-legend that is obtained through the `legend.loc` in `plot` as demonstrated above and simply usees the column names of the xts object. The function is called `addLegend()`

— note that you need to specify the argument `lwd` (line width) if you want a line drawn next to the legend names!

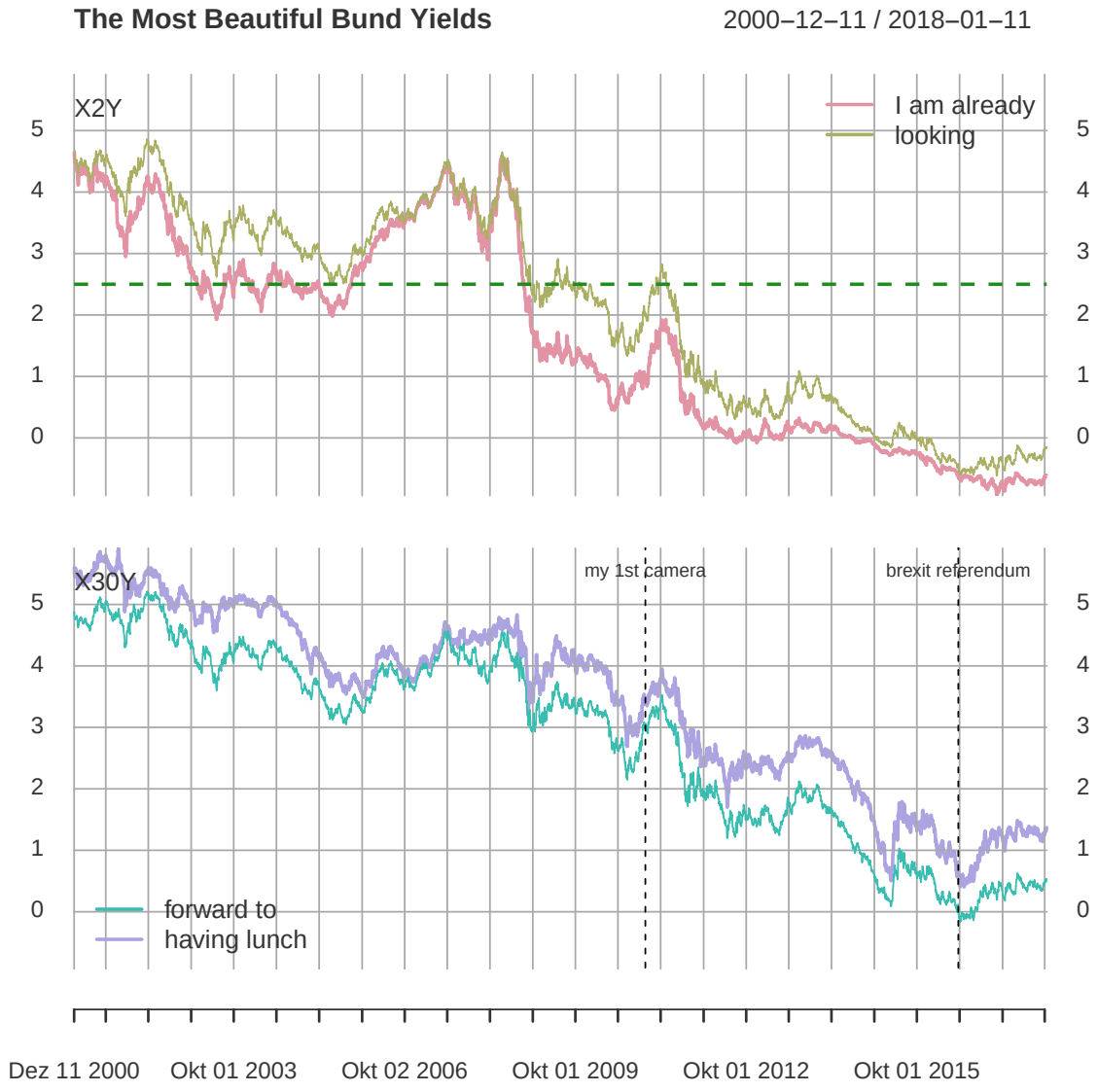
```
#note: you don't need to assign to a new object (myplot)
#if you don't, you'll also get plots of the intermediary steps
myplot <- plot(d[,c(1,4)],
               main="The Most Beautiful Bund Yields",
               col=rainbow_hcl(4)[c(1,4)],
               multi.panel=2)
myplot <- lines(d[,2], on=1, col=rainbow_hcl(4)[2])
myplot <- lines(d[,3], on=2, col=rainbow_hcl(4)[3])
myplot <- addLegend("topright",
                    legend.names=c("I am already","looking"),
                    col=rainbow_hcl(4)[1:2],
                    lwd=2, on=1)
myplot <- addLegend("bottomleft",
                    legend.names=c("forward to","having lunch"),
                    col=rainbow_hcl(4)[3:4],
                    lwd=2, on=2)
myplot
```



If you want to add horizontal lines to a plot, it is best to create an additional timeseries of constant values and add this to your plot using `lines()`. For vertical lines, use `addEventLines()`. The main argument of the function is called `events`, which is a

character vector converted to class `xts`. The dates of this `xts` object indicate where to draw the vertical lines, while the character strings indicate the labelling of the vertical lines. Again, for multi-panel plots, use `on` to indicate the panel to draw on.

```
# adding a horizontal line at height 2.5 to the first panel
# in our plot from above
t_stamps <- time(d) # getting time stamps of our data
hline <- xts(rep(2.5, length(t_stamps)) , order.by=t_stamps)
myplot <- lines(hline, on=1,
                lty=2, lwd=2, # line type and line width
                col="forestgreen")


# adding two vertical lines to the second panel
myevents <- xts(c("my 1st camera", "brexit referendum"),
                order.by=as.Date(c("2010-12-24", "2016-06-23")) )
myplot <- addEventLines(events=myevents, on=2,
                        lty=2, pos=1,
                        cex=0.7) # adjusts size of text
#pos to adjust the defaul position (1= below, 2=to the left, 3=above,...)
#additionally, use srt=90 to rotate the text by 90 degrees

myplot
```



For any additional infos on plotting `xts` objects, type `?plot.xts`!

3.3. Handling `xts` objects

3.3.1. Timeindex, subsetting time, lags and differences

To access the time index associated to an `xts` object, use `time()`, which will return an object of class `Date`.

```
head(time(d))

## [1] "2000-12-11" "2000-12-12" "2000-12-13" "2000-12-14" "2000-12-15"
## [6] "2000-12-18"
```

Note that for `Date` objects, you can easily get previous (later) dates by addition (substraction) of an integer, you can calculate the time difference (in calendar days!) between two dates and you can use `seq()` to create a sequence of all dates between to timepoints in the `Date`-format:

```
(mydates <- time(d)[1:10])

##  [1] "2000-12-11" "2000-12-12" "2000-12-13" "2000-12-14" "2000-12-15"
##  [6] "2000-12-18" "2000-12-19" "2000-12-20" "2000-12-21" "2000-12-22"

mydates - 3

##  [1] "2000-12-08" "2000-12-09" "2000-12-10" "2000-12-11" "2000-12-12"
##  [6] "2000-12-15" "2000-12-16" "2000-12-17" "2000-12-18" "2000-12-19"

mydates + 1

##  [1] "2000-12-12" "2000-12-13" "2000-12-14" "2000-12-15" "2000-12-16"
##  [6] "2000-12-19" "2000-12-20" "2000-12-21" "2000-12-22" "2000-12-23"

mydates[10] - mydates[1]

## Time difference of 11 days

seq(from=as.Date("1994-08-11"), to=as.Date("1994-08-15"), by=1)

## [1] "1994-08-11" "1994-08-12" "1994-08-13" "1994-08-14" "1994-08-15"
```

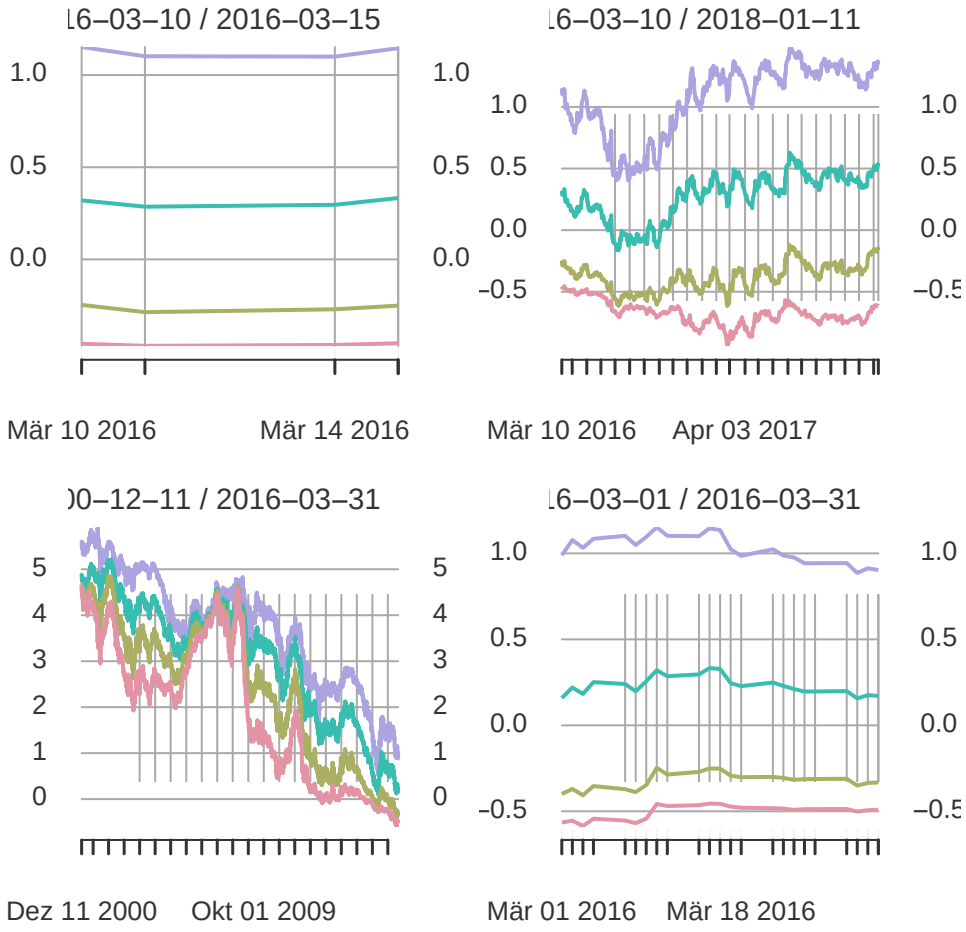
Use `weekdays()` to receive the day of the week associated to a particular date:

```
weekdays(mydates)

##  [1] "Montag"     "Dienstag"   "Mittwoch"   "Donnerstag" "Freitag"
##  [6] "Montag"     "Dienstag"   "Mittwoch"   "Donnerstag" "Freitag"
```

`xts` allows you to quickly subset to a specified time period:

```
d["2016-03-10"] #one particular date
par(mfrow=c(2,2)) # to plot four windows
plot(d["2016-03-10/2016-03-15"], #all dates in this range
     col=rainbow_hcl(4), main="")
plot(d["2016-03-10/"], # all dates from
     col=rainbow_hcl(4), main="")
plot(d["/2016-03"], # all dates up to
     col=rainbow_hcl(4), main="")
plot(d["2016-03"], # particular month (similar for year)
     col=rainbow_hcl(4), main="")
par(mfrow=c(1,1)) #reset to plotting just one window
```

```
##                  X2Y      X5Y    X10Y    X30Y
## 2016-03-10 -0.45785 -0.24702 0.32032 1.15206
```



To access the first and the last observatoins of a timeseries, use `first()` and `last()`. For example:

```
# first two weeks of the data
first(d, "2 weeks")

##                  X2Y      X5Y    X10Y    X30Y
## 2000-12-11 4.65575 4.67264 4.85954 5.58972
## 2000-12-12 4.65641 4.67391 4.88013 5.59810
## 2000-12-13 4.59706 4.60987 4.83391 5.55546
## 2000-12-14 4.52651 4.57871 4.81444 5.48557
## 2000-12-15 4.50157 4.52840 4.77875 5.45185
## 2000-12-18 4.47767 4.48716 4.75643 5.46294
## 2000-12-19 4.51193 4.52899 4.79007 5.53795
## 2000-12-20 4.44878 4.48937 4.75721 5.58982
## 2000-12-21 4.43995 4.51261 4.77729 5.56935
## 2000-12-22 4.42820 4.51738 4.78127 5.53622

# first three days of the fourth week
first(last(first(d, "4 weeks"), "1 week"), "3 days")
```

```
##                 X2Y     X5Y    X10Y    X30Y
## 2001-01-01 4.33708 4.44254 4.76785 5.44204
## 2001-01-02 4.23014 4.33337 4.66470 5.37103
## 2001-01-03 4.24252 4.34597 4.63979 5.31879
```

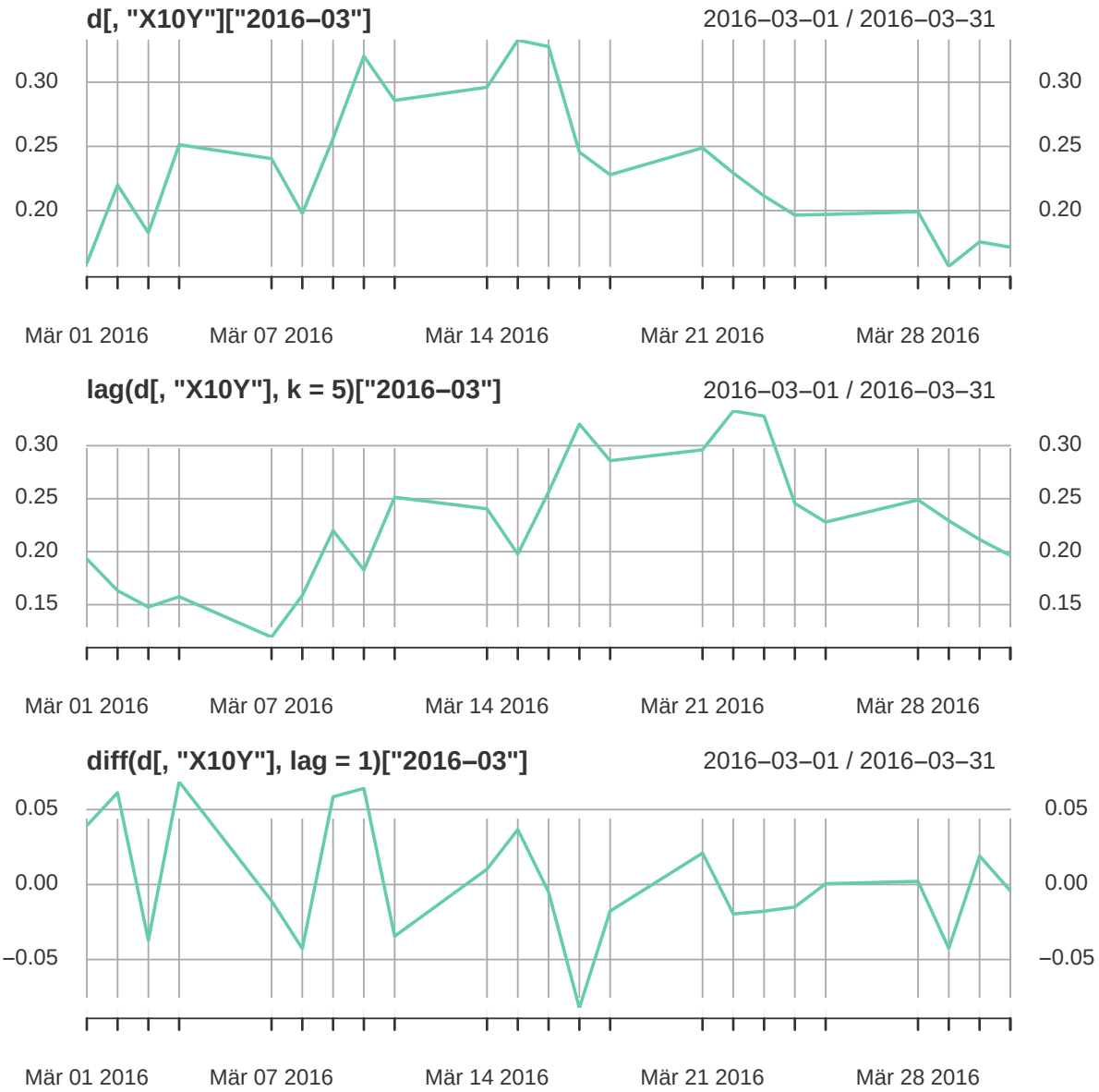
To access the last datapoint of each week/month/year, use `endpoints()`. This will give you the index indicating the relevant datapoints (or rows if your `xts` object has multiple columns).

```
#index indicating last row of each year
endpoints(d, on="years")

##  [1]    0   15  276  537  798 1060 1320 1580 1841 2103 2364 2625 2885 3146 3407
## [16] 3668 3929 4190 4450 4459

d[endpoints(d, on="years"),] #last row of each year

##                 X2Y      X5Y    X10Y    X30Y
## 2000-12-29  4.33923  4.45311 4.76962 5.45007
## 2001-12-31  3.59524  4.41354 5.00067 5.42061
## 2002-12-31  2.69079  3.45817 4.29335 4.91846
## 2003-12-31  2.57723  3.53304 4.29768 5.03775
## 2004-12-31  2.44981  3.04501 3.68515 4.39303
## 2005-12-30  2.81149  3.03830 3.27411 3.55144
## 2006-12-29  3.80748  3.84121 3.86522 3.95884
## 2007-12-31  3.90691  4.05016 4.27173 4.50824
## 2008-12-31  1.73927  2.36507 2.96356 3.46194
## 2009-12-31  1.33862  2.49626 3.42280 4.16943
## 2010-12-31  0.86843  1.93991 2.99559 3.42105
## 2011-12-30  0.14744  0.83049 1.86695 2.37195
## 2012-12-31 -0.02049  0.35135 1.41290 2.24353
## 2013-12-31  0.21501  0.97978 2.04641 2.81223
## 2014-12-31 -0.10896  0.02032 0.60421 1.46853
## 2015-12-31 -0.34853 -0.02099 0.70123 1.57225
## 2016-12-30 -0.80623 -0.50247 0.24764 1.04166
## 2017-12-29 -0.63731 -0.17247 0.47965 1.30400
## 2018-01-11 -0.61599 -0.16107 0.51724 1.34517

#first row of each year
d[endpoints(d, on="years")[-length(endpoints(d, on="years"))] +1, ]

##                 X2Y      X5Y    X10Y    X30Y
## 2000-12-11  4.65575  4.67264 4.85954 5.58972
## 2001-01-01  4.33708  4.44254 4.76785 5.44204
## 2002-01-01  3.59154  4.41874 5.00021 5.42015
## 2003-01-01  2.69262  3.46202 4.29397 4.91920
## 2004-01-01  2.57856  3.53289 4.29947 5.03928
## 2005-01-03  2.44139  3.01346 3.64213 4.35107
## 2006-01-02  2.84999  3.06794 3.29576 3.56884
```

```
## 2007-01-01  3.79818  3.83792 3.86028 3.97272
## 2008-01-01  3.90779  4.04984 4.26674 4.50876
## 2009-01-01  1.74305  2.36545 2.96350 3.46514
## 2010-01-01  1.34098  2.49941 3.42393 4.17089
## 2011-01-03  0.83419  1.92101 2.95524 3.36222
## 2012-01-02  0.20980  0.89541 1.94642 2.45610
## 2013-01-01 -0.02051  0.35154 1.41329 2.24374
## 2014-01-01  0.21738  0.98178 2.04698 2.81230
## 2015-01-01 -0.10889  0.02072 0.60514 1.46868
## 2016-01-01 -0.34804 -0.01928 0.70239 1.57246
## 2017-01-02 -0.79844 -0.51094 0.23256 1.02898
## 2018-01-01 -0.63457 -0.16940 0.48261 1.30474

#first row of each year even simpler
#using the unexported function startof()
d[xts:::startof(d, by="years")]

##                 X2Y      X5Y    X10Y    X30Y
## 2000-12-11  4.65575  4.67264 4.85954 5.58972
## 2001-01-01  4.33708  4.44254 4.76785 5.44204
## 2002-01-01  3.59154  4.41874 5.00021 5.42015
## 2003-01-01  2.69262  3.46202 4.29397 4.91920
## 2004-01-01  2.57856  3.53289 4.29947 5.03928
## 2005-01-03  2.44139  3.01346 3.64213 4.35107
## 2006-01-02  2.84999  3.06794 3.29576 3.56884
## 2007-01-01  3.79818  3.83792 3.86028 3.97272
## 2008-01-01  3.90779  4.04984 4.26674 4.50876
## 2009-01-01  1.74305  2.36545 2.96350 3.46514
## 2010-01-01  1.34098  2.49941 3.42393 4.17089
## 2011-01-03  0.83419  1.92101 2.95524 3.36222
## 2012-01-02  0.20980  0.89541 1.94642 2.45610
## 2013-01-01 -0.02051  0.35154 1.41329 2.24374
## 2014-01-01  0.21738  0.98178 2.04698 2.81230
## 2015-01-01 -0.10889  0.02072 0.60514 1.46868
## 2016-01-01 -0.34804 -0.01928 0.70239 1.57246
## 2017-01-02 -0.79844 -0.51094 0.23256 1.02898
## 2018-01-01 -0.63457 -0.16940 0.48261 1.30474
```

To access timelags of an `xts` timeseries or calculate first differences, use `lag()` and `diff()`.

```r
par(mfrow=c(3,1))
#taking only the 10Y yield and subsetting to month March of 2016
plot(d[,"X10Y"]["2016-03"],
     col="mediumaquamarine") #original timeseries
plot(lag(d[,"X10Y"], k=5)["2016-03"],
     col="mediumaquamarine") #lagged by 5 trading days (one week)
plot(diff(d[,"X10Y"],lag=1)["2016-03"],
     col="mediumaquamarine") #difference to previous day
```



3.3.2. ATTENTION!!!

You can combine two `xts` objects with different dates into one matrix and `xts` will handle this for you automatically[1]:

[1] If you wish to carry over previous/next values to fill missing observations, use `na.locf()` (we use it below in section 3.3.3 on `apply.weekly`)

```r
(ts1 <- d[1:5, 1])
```

```
##                X2Y
## 2000-12-11 4.65575
## 2000-12-12 4.65641
## 2000-12-13 4.59706
## 2000-12-14 4.52651
## 2000-12-15 4.50157
```

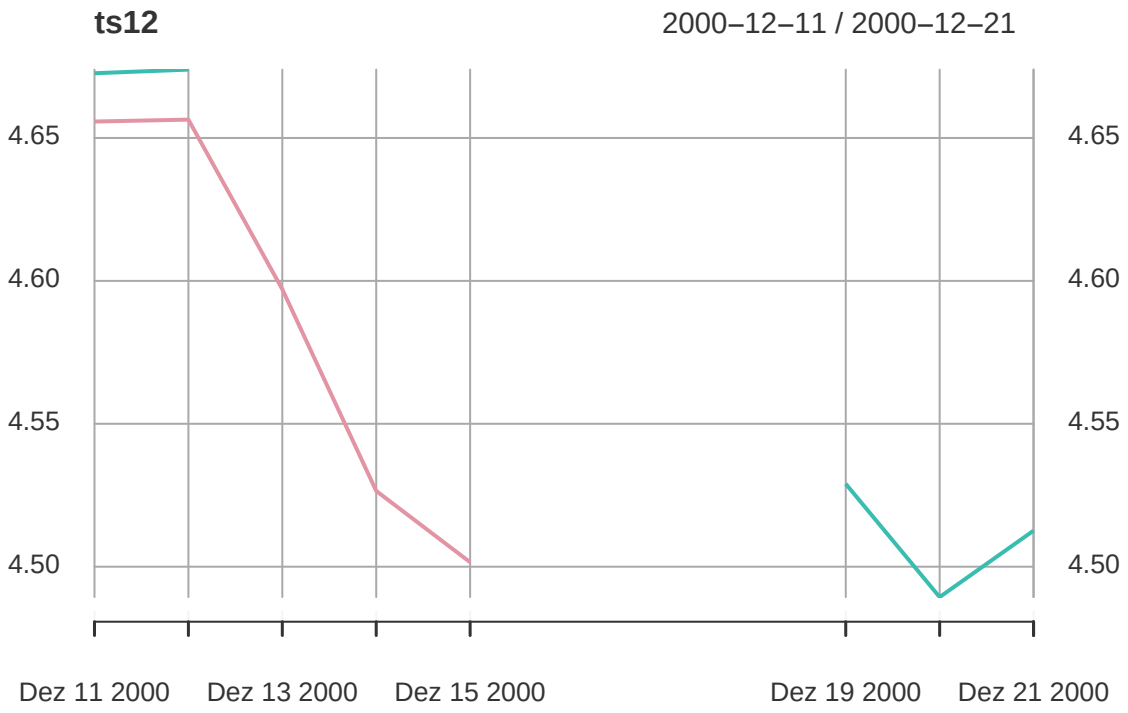
```r
(ts2 <- d[c(1:2,7:9), 2])
```

```
##                X5Y
## 2000-12-11 4.67264
## 2000-12-12 4.67391
## 2000-12-19 4.52899
## 2000-12-20 4.48937
## 2000-12-21 4.51261
```

```r
(ts12 <- cbind(ts1, ts2))
```

```
##                X2Y     X5Y
## 2000-12-11 4.65575 4.67264
## 2000-12-12 4.65641 4.67391
## 2000-12-13 4.59706      NA
## 2000-12-14 4.52651      NA
## 2000-12-15 4.50157      NA
## 2000-12-19      NA 4.52899
## 2000-12-20      NA 4.48937
## 2000-12-21      NA 4.51261
```

```r
plot(ts12, col=rainbow_hcl(2), legen.loc="topright")
```



However, observe the following:

```
#d1 starts one day earlier and ends one day earlier than d2
(d1 <- d[1:5,1])

##                X2Y
## 2000-12-11 4.65575
## 2000-12-12 4.65641
## 2000-12-13 4.59706
## 2000-12-14 4.52651
## 2000-12-15 4.50157

(d2 <- d[2:6,1])

##                X2Y
## 2000-12-12 4.65641
## 2000-12-13 4.59706
## 2000-12-14 4.52651
## 2000-12-15 4.50157
## 2000-12-18 4.47767

#first differences by taking d2-d1
d2-d1

##            X2Y
## 2000-12-12   0
## 2000-12-13   0
## 2000-12-14   0
## 2000-12-15   0
```

As you can see, when performing arithmetic operations, `xts` a) matches dates and b) truncates the timeseries to the dates present in both series!

Solution: Convert at least one of the two `xts` objects back into a "normal" vector or matrix using `coredata()`.

```
d2-coredata(d1) # again xts since d2 is still in xts format

##                  X2Y
## 2000-12-12  0.00066
## 2000-12-13 -0.05935
## 2000-12-14 -0.07055
## 2000-12-15 -0.02494
## 2000-12-18 -0.02390

# coredata(d1) is still in matrix form because it was previously
# taken as one column of a matrix:
coredata(d1)

##          X2Y
## [1,] 4.65575
## [2,] 4.65641
## [3,] 4.59706
## [4,] 4.52651
## [5,] 4.50157
```

3.3.3. `apply.weekly()`

Often, you want to perform certain calculations on each week, month or year of a timeseries. If observations are on irregular time intervals – e.g. one week with 5 trading days, but the next one with only three trading days, then a week with four trading days and finally two more weeks with 5 trading days – it can be very tedious to get the indexing right when trying to do something with the data of each week. Conveniently, `xts` provides the functions `apply.weekly`, `apply.monthly` and `apply.yearly`. In the following, the concept will be demonstrated using `apply.weekly`. `apply.weekly` basically requires two arguments: a) the `xts` object you want to apply it to, b) the argument `FUN`, which specifies the function you want to apply to each week

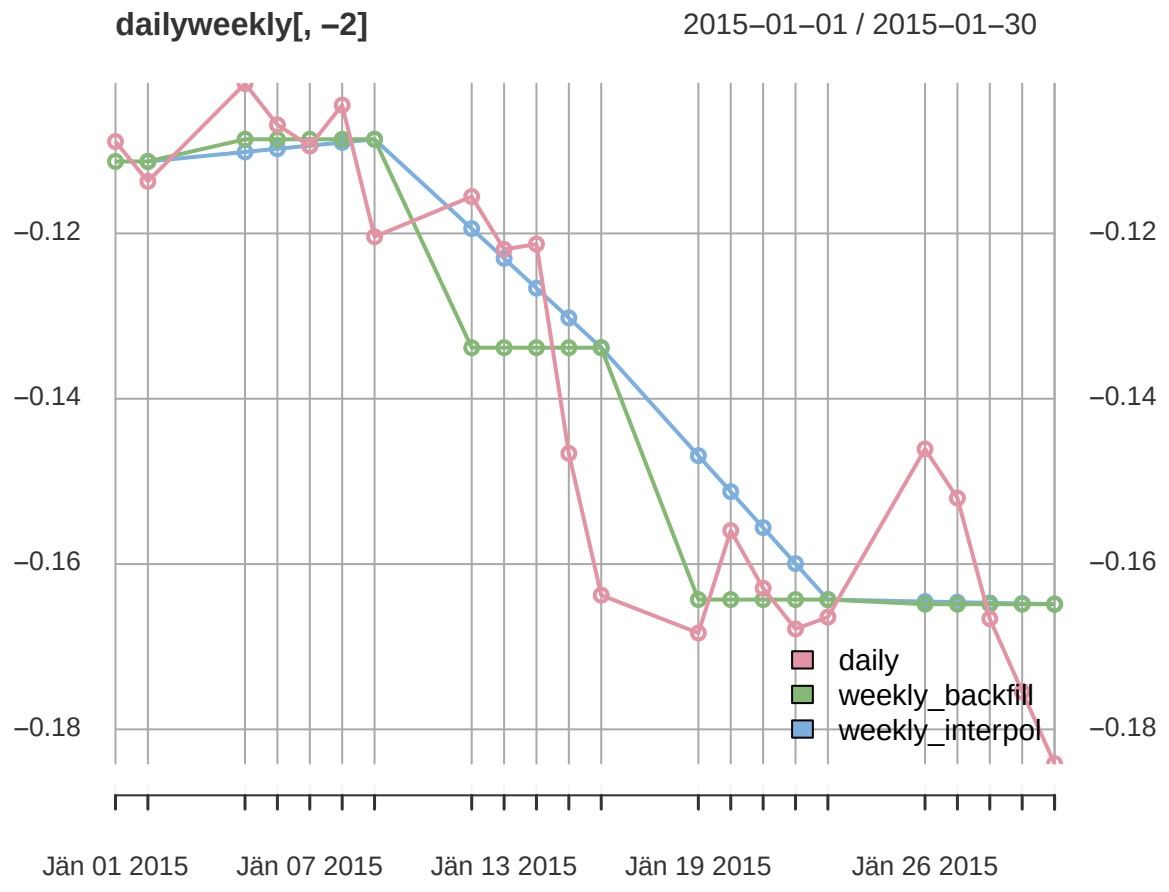
Let's start with a univariate timeseries:

```
# 2Y yields in January, 2016
d1 <- d[,"X2Y"]["2015-01"]

# computing average yield of each week
(d1_weekly <- apply.weekly(d1, FUN=function(x) sum(x)/length(x) ))

##                  X2Y
## 2015-01-02 -0.111295
## 2015-01-09 -0.108610
## 2015-01-16 -0.133826
```

```
## 2015-01-23 -0.164298
## 2015-01-30 -0.164856

# you can disregard the warning message
dailyweekly <- (cbind(d1, d1_weekly))
colnames(dailyweekly) <- c("daily", "weekly")
head(dailyweekly)

##                daily    weekly
## 2015-01-01 -0.10889        NA
## 2015-01-02 -0.11370 -0.111295
## 2015-01-05 -0.10188        NA
## 2015-01-06 -0.10686        NA
## 2015-01-07 -0.10943        NA
## 2015-01-08 -0.10448        NA
```

Weekly averages are obviously at a lower frquency (always assigned to last day of week) compared to daily data. To plot the two together, you need to fill the missing values somehow. Missing values can be filled by using `na.locf()`. Setting the argument `fromLast=TRUE` will fill missing values with the *next* available value, otherwise the closest *previous* available value is used. Alternatively, using `na.approx()`, missing values can be filled by linearly interpolating between the next and the previous available observation. Both `na.locf()` and `na.approx()` are also useful on many applications other than plotting where you want to fill up missing values.

```r
dailyweekly <- cbind(dailyweekly, na.locf(dailyweekly[,2], fromLast=TRUE))
dailyweekly <- cbind(dailyweekly, na.approx(dailyweekly[,2]) )
colnames(dailyweekly) <- c("daily", "weekly", "weekly_backfill",
                           "weekly_interpol")
head(dailyweekly) #first date obviously cannot be filled by interpolation

##                daily    weekly weekly_backfill weekly_interpol
## 2015-01-01 -0.10889        NA       -0.111295              NA
## 2015-01-02 -0.11370 -0.111295       -0.111295      -0.1112950
## 2015-01-05 -0.10188        NA       -0.108610      -0.1101443
## 2015-01-06 -0.10686        NA       -0.108610      -0.1097607
## 2015-01-07 -0.10943        NA       -0.108610      -0.1093771
## 2015-01-08 -0.10448        NA       -0.108610      -0.1089936

plot(dailyweekly[,-2], col=rainbow_hcl(3),
     legend.loc="bottomright", type="o")
```



Next, let's have a look at `apply.weekly()` for an `xts` object with multiple columns, such as `d`. In this case, `apply.weekly()` will apply the function specified by the argument `FUN` to every week, but *across all columns* instead of separately for each column! See for yourself:

```
# computing weekly averages
weekly <- apply.weekly(d, FUN=function(x) sum(x)/length(x) )
head(weekly)

##                 [,1]
## 2000-12-15 4.892415
## 2000-12-22 4.820029
## 2000-12-29 4.794783
## 2001-01-05 4.656582
## 2001-01-12 4.695222
## 2001-01-19 4.766852
```

Indeed, only one timeseries is left, whereas we wanted to have the weekly averages of the four series separately. That's quite sad...

SOLUTION: The function provided via the argument `FUN` must be specified such that once it is passed all columns of a particular week by `apply.weekly`, it applies its operations columnwise! The most general way to do this is to use `apply(A,margin=2,FUN)`. This applies the function specified in `FUN` to each row (if `margin=1`) or column (if `margin=2`, the case we need here) of the matrix `A`. Consider the following example:

```
(A <- matrix(1:9, ncol=3))

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

apply(A,1,FUN=function(x) sum(x) ) #sums each row

## [1] 12 15 18

apply(A,2, FUN=function(x) sum(x)) #sums each column

## [1]  6 15 24
```

Now let's use this in our initial problem of computing weekly averages:

```
# computing weekly averages
weekly <- apply.weekly(d, FUN=function(A){
  apply(A,2,FUN=function(x) sum(x)/length(x))
})
head(weekly)

##                 X2Y      X5Y     X10Y     X30Y
## 2000-12-15 4.587460 4.612706 4.833354 5.536140
## 2000-12-22 4.461306 4.507102 4.772454 5.539256
## 2000-12-29 4.395288 4.496380 4.783874 5.503588
## 2001-01-05 4.222490 4.354830 4.669968 5.379040
## 2001-01-12 4.255180 4.403108 4.687724 5.434878
## 2001-01-19 4.335422 4.471360 4.751152 5.509474
```

```
# It works!
```

4. From stocks to portfolios

4.1. Aggregating returns

When talking about the aggregation of stock returns through time and across the cross-section it is very important to clarify whether we use simple returns or log-returns. This section intends to show you the differences when handling both types of return.

Let us start with the price data.

```
#load the price data set, already cleaned and converted to xts
load("../data/prices.RData")
head(price[,1:5])

##              X10001   X10002    X10025   X10026    X10028
## 1990-01-31 3.311181 4.181425 10.474186 5.306980 0.4693490
## 1990-02-28 3.284937 4.245365 10.226498 4.442308 0.4871051
## 1990-03-30 3.287699 4.256135  9.769611 4.294397 0.3802039
## 1990-04-30 3.300391 4.237388  9.434155 4.506959 0.3308849
## 1990-05-31 3.243311 4.247749  9.952157 4.339328 0.2455050
## 1990-06-29 3.259194 4.239016  9.563001 4.077300 0.2307414
```

Next we compute both simple and log-returns.

```
##compute simple returns
simple.r <- (price/lag(price)-1) [-1 ,] #1st row is NA, so we remove it
head(simple.r[,1:5])

##                  X10001       X10002      X10025      X10026      X10028
## 1990-02-28 -0.0079257711  0.015291621 -0.02364745 -0.16293110  0.03783142
## 1990-03-30  0.0008407991  0.002536739 -0.04467678 -0.03329589 -0.21946226
## 1990-04-30  0.0038604096 -0.004404698 -0.03433676  0.04949759 -0.12971733
## 1990-05-31 -0.0172948928  0.002445319  0.05490717 -0.03719380 -0.25803494
## 1990-06-29  0.0048971297 -0.002055966 -0.03910267 -0.06038468 -0.06013560
## 1990-07-31  0.0215810735 -0.034443092 -0.02627608  0.03643682  0.19590807

##compute log returns
log.r <- diff(log(price))[-1,]
head(log.r[,1:5])

##                  X10001       X10002      X10025      X10026      X10028
## 1990-02-28 -0.0079573470  0.015175883 -0.02393154 -0.17784889  0.03713336
## 1990-03-30  0.0008404458  0.002533527 -0.04570554 -0.03386281 -0.24777218
## 1990-04-30  0.0038529774 -0.004414428 -0.03494012  0.04831157 -0.13893722
## 1990-05-31 -0.0174461965  0.002442334  0.05345277 -0.03790313 -0.29845313
## 1990-06-29  0.0048851778 -0.002058082 -0.03988771 -0.06228472 -0.06201967
## 1990-07-31  0.0213514992 -0.035050238 -0.02662746  0.03578869  0.17890579
```

We can always switch between simple (r_t) and log-returns (r_t^l) because of the following

relation:

$$
\begin{aligned}
r_t &= \frac{p_t - p_{t-1}}{p_{t-1}} = \frac{p_t}{p_{t-1}} - 1 \\
1 + r_t &= \frac{p_t}{p_{t-1}} \\
r_t^l &= \log(1 + r_t) = \log\left(\frac{p_t}{p_{t-1}}\right) = \log\left(p_t\right) - \log\left(p_{t-1}\right) \\
&\implies r_t = e^{r_t^l} - 1. \qquad (1)
\end{aligned}
$$

Note that the two returns are not quite the same, but are very close to each other. Actually, the smaller the change in prices the closer simple and log-returns are to each other. This is also called the *approximate raw-log equality*[2], which formally reads

$$\log\left(1 + r\right) \approx r \ll 1.$$

```
prices_tmp <- c(1,1.01,1.08,1.15,1.10,2.5,5.6,20.1)
container <- data.frame("prices"=prices_tmp,
                        "simple returns"=c(NA,prices_tmp[-1]/prices_tmp[-8]-1),
                        "log returns"=c(NA,diff(log(prices_tmp))))
container
```

```
##   prices simple.returns  log.returns
## 1   1.00             NA           NA
## 2   1.01     0.01000000  0.009950331
## 3   1.08     0.06930693  0.067010710
## 4   1.15     0.06481481  0.062800901
## 5   1.10    -0.04347826 -0.044451763
## 6   2.50     1.27272727  0.820980552
## 7   5.60     1.24000000  0.806475866
## 8  20.10     2.58928571  1.277953217
```

Aggregation through time

The benefit of log-returns is their time additivity. In contrast to simple returns we do not need to compound the returns by multiplication but can simply add them up. This is again because of (1). Consider four quarterly returns, then the return after one year is:

$$
\begin{aligned}
(1 + r_Y) &= (1 + r_{Q1})(1 + r_{Q2})(1 + r_{Q3})(1 + r_{Q4}) \\
\text{using (1)} \quad (1 + r_Y) &= (1 + e^{r_{Q1}^l} - 1)(1 + e^{r_{Q2}^l} - 1)(1 + e^{r_{Q3}^l} - 1)(1 + e^{r_{Q4}^l} - 1) \\
&= e^{r_{Q1}^l + r_{Q2}^l + r_{Q3}^l + r_{Q4}^l} = e^{r_Y^l} \\
&\implies r_Y^l = r_{Q1}^l + r_{Q2}^l + r_{Q3}^l + r_{Q4}^l
\end{aligned}
$$

We can check this result with R.

[2]Simple returns are often also called raw returns.

```
aggregated_returns <- cbind(cumprod(1+simple.r["2000",1])-1,
                            cumsum(log.r["2000",1]),
                            exp(cumsum(log.r["2000",1]))-1)
names(aggregated_returns) <- c("simple", "log", "exp(log)-1")
aggregated_returns
```

```
##                  simple         log  exp(log)-1
## 2000-01-31 -0.04710992 -0.04825572 -0.04710992
## 2000-02-29 -0.03010830 -0.03057086 -0.03010830
## 2000-03-31 -0.06247066 -0.06450722 -0.06247066
## 2000-04-28 -0.04806661 -0.04926022 -0.04806661
## 2000-05-31 -0.07252369 -0.07528802 -0.07252369
## 2000-06-30 -0.05910427 -0.06092296 -0.05910427
## 2000-07-31 -0.07405206 -0.07693727 -0.07405206
## 2000-08-31 -0.02916421 -0.02959794 -0.02916421
## 2000-09-29  0.02663153  0.02628309  0.02663153
## 2000-10-31  0.05757552  0.05597905  0.05757552
## 2000-11-30  0.12437856  0.11723049  0.12437856
## 2000-12-29  0.14659193  0.13679400  0.14659193
```

This comes in very handy when calculating the average monthly return over the year 2000 for example. We can use the arithmetic mean for log-returns, but for simple returns we need to use the geometric mean. If we use the arithmetic mean for simple returns, we overestimate the monthly average. On the other hand, if we use the geometric mean for log-returns, we underestimate the montly average.

```
##geometric mean of simple returns
gm_simple.r <- prod(1+simple.r["2000",1])^(1/12)-1
gm_simple.r
```

```
## [1] 0.01146472
```

```
##arithmetic mean of log returns
am_log.r <- mean(log.r["2000",1])
am_log.r
```

```
## [1] 0.0113995
```

```
##are they the "same"?
gm_simple.r - (exp(am_log.r)-1) #yes!
```

```
## [1] 0
```

```
##the arithmetic mean of simple returns
am_simple.r <- mean(simple.r["2000",1])
am_simple.r
```

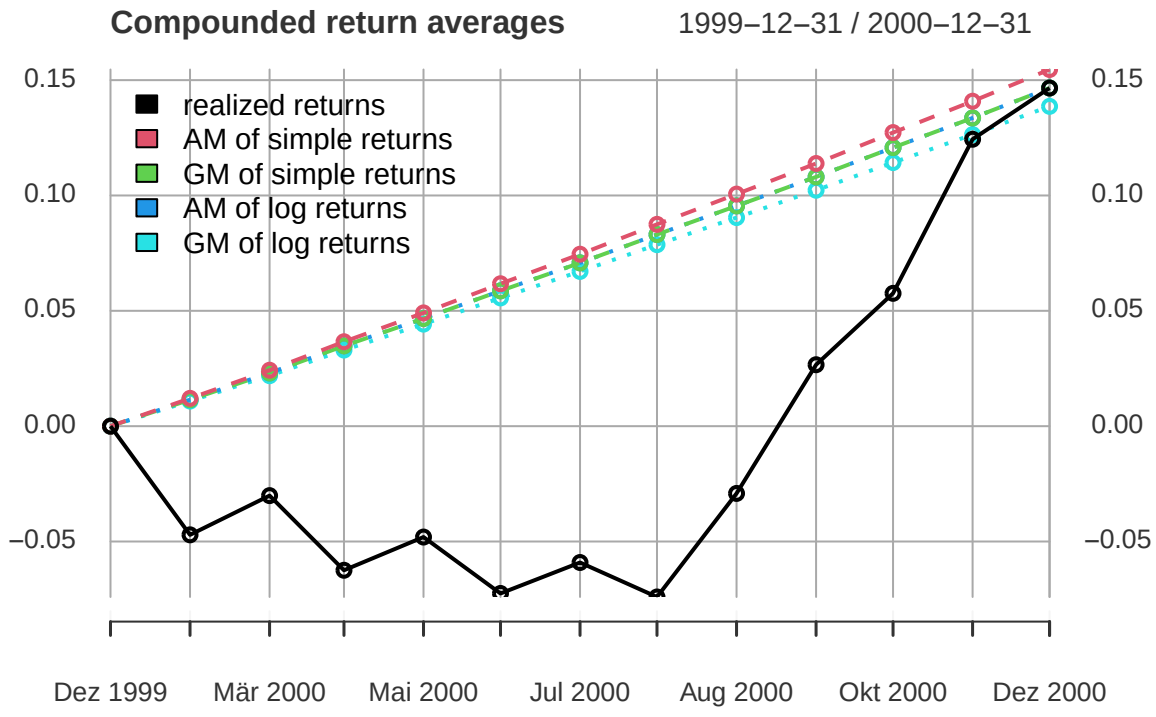
```
## [1] 0.01205269
```

```
##geometric mean of log returns
gm_log.r <- prod(1+log.r["2000",1])^(1/12)-1
gm_log.r
```

```
## [1] 0.01081976
```

Indeed, the geometric mean of the log returns underestimates and the arithmetic mean of the simple returns overestimates the true monthly average. We can also visualize this quite nicely. In the following figure, we first compute both the arithmetic and geometric means for simple as well as log returns. Using these averages, we then compute the cumulative return up to time t for each point in time over our time window. For simple returns, as usual, the cumulative return up to time t is calculated as $(1+\bar{r})^t - 1$, with $\bar{r}$ representing either the arithmetic or the geometric average. For log returns, cumulative returns (converted back to simple returns) are calculated as $e^{\bar{r}t} - 1$.

```
##the compounded return time series
container <- c(1,cumprod(1+simple.r["2000",1]))-1
##using the AM&GM of simple and log returns as averages
container <- xts(cbind(container,
                       c(1,cumprod(1+rep(am_simple.r,12)))-1,
                       c(1,cumprod(1+rep(gm_simple.r,12)))-1,
                       exp(c(0,cumsum(rep(am_log.r,12))))-1,
                       exp(c(0,cumsum(rep(gm_log.r,12))))-1),
                 order.by=seq(as.Date("1999-12-31"),
                              as.Date("2000-12-31"),
                              by="month"))

names(container) <- c("realized returns","AM of simple returns",
                      "GM of simple returns","AM of log returns",
                      "GM of log returns")
plot(container, type="o",legend.loc="topleft", lty=c(1,2,2,3,3),
     main = "Compounded return averages")
```



Aggregation in the cross-section

Aggregation in the cross-section is needed when computing portfolio returns, i.e. the overall return across all assets. For example, you hold two stocks with returns $r_A = 3\%$ and $r_B = 5\%$, with the returns measured from yesterday to today. Yesterday stock A was worth EUR 100 and stock B EUR 50, so the portfolio weights are $w_A = 2/3$ and $w_B = 1/3$. What is the return of your portfolio, so the return of stocks A and B combined?

Let us look at this step-by-step. Your total stock wealth yesterday was 100+50=150. Your total stock wealth today is

```
100*1.03+50*1.05

## [1] 155.5
```

Thus your portfolio return is $(155.5/150) - 1 \approx 3.67\%$. Ideally, we can come up with a clever way that circumvents the steps of computing the portfolio/stock wealth every time and simply uses the returns and portfolio weights to compute the portfolio return. Luckily, we can simply use the weighted sum of stock returns to compute the portfolio return.

```
(2/3)*0.03+(1/3)*0.05

## [1] 0.03666667
```

Why is that the case? Let v_x denote the value of stock x (at the beginning of the period over which we want to compute the return) and recall that, taking stock A as an example, the portfolio weight is computed as

$$w_A = \frac{v_A}{v_A + v_B}$$

Then

$$\begin{aligned}
(v_A + v_B)(1 + r_{PF}) &= v_A(1 + r_A) + v_B(1 + r_B) \\
(1 + r_{PF}) &= \frac{v_A}{v_A + v_B}(1 + r_A) + \frac{v_B}{v_A + v_B}(1 + r_B) \\
&= \frac{v_A}{v_A + v_B} + \frac{v_B}{v_A + v_B} + \frac{v_A}{v_A + v_B} r_A + \frac{v_B}{v_A + v_B} r_B \\
&= 1 + \frac{v_A}{v_A + v_B} r_A + \frac{v_A}{v_A + v_B} r_B \\
&= 1 + w_A r_A + w_B r_B
\end{aligned}$$

by the definiton of w_x we therefore have $r_{PF} = w_A r_A + w_B r_B$.

There is no way to compute the portfolio return using log-returns, we can only approximate portfolio returns using using the weighted sum of log-returns. Therefore, we always need to transform log-returns to simple returns before calculating the portfolio return.

Let us look at another example of cross-sectional return aggregation. Practitioners as well as academics in the field of finance are often interested in the so called *market return*. In theory, the *market return* is the value weighted return of all companies, but

in reality we cannot observe the change in assets of every company. Thus, one usually computes the *market return* by computing the return of all companies listed on a stock exchange (omitting those companies not listed) weighted by their market capitalization. This aggregate number is very important for several reasons as you will learn in this course. For instance, it indicates if the economy as a whole appreciated or not during the observation period.

To compute the market return in t we need the market capitalization in $t-1$. Why $t-1$ and not t? Because the market return in t is the change in prices from $t-1$ to t. If we would use the weights from t, then these weights would already incorporate the price change from $t-1$ to t and thus not represent the value weighted change in prices. The following example should make it clear how to compute the *market return* for a given set of stocks.

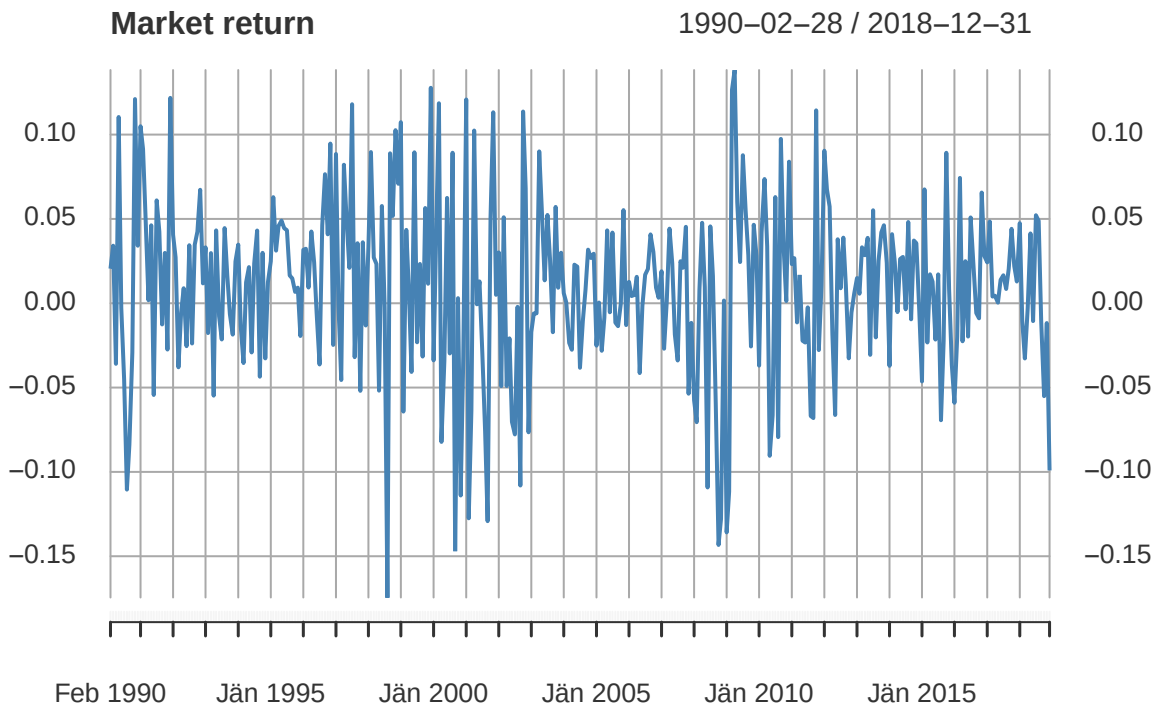
```
head(price_data)
```

```
##   X    ID       Name       Date    Price Shares.Outstanding
## 1 1 10001 Dictum Ltd 1986-01-31 2.055802                2955
## 2 2 10001 Dictum Ltd 1986-02-28 2.091438                2955
## 3 3 10001 Dictum Ltd 1986-03-31 2.098600                2955
## 4 4 10001 Dictum Ltd 1986-04-30 2.128636                2955
## 5 5 10001 Dictum Ltd 1986-05-30 2.106847                2955
## 6 6 10001 Dictum Ltd 1986-06-30 2.047738                2955
```

```
## the market capitalization is price * shares outstanding
price_data$mcap <- abs(price_data$Price) * price_data$Shares.Outstanding
## in this dataset negative prices indicate that there was no closing price
## available and the bid-ask average is used instead
IDs <- unique(price_data$ID)
## extracting unique firm IDs included in the dataset
## create an empty list: each list element will
## store the whole xts time series
## of mcaps corresponding to one particular firm
## (identified through the ID)
mcap <- lapply(X = IDs, FUN = function(x){
  xts(x = price_data[price_data$ID==x,7], #the mcap is stored column 7
      order.by = as.Date(as.character(price_data[price_data$ID==x,4])))
  ## the date in column 4
})
names(mcap)<-paste("X",unique(price_data$ID)) #name the xts columns
prices <- lapply(X = IDs, FUN = function(x){
  xts(x = abs(price_data[price_data$ID==x,5]), #the price is stored column 5
      order.by = as.Date(as.character(price_data[price_data$ID==x,4])))
  ## the date in column 4
})
names(prices) <- paste("X",unique(price_data$ID)) #name the xts columns

## cbinding all list-elements together using do.call
```

```r
mcap <- do.call(cbind, mcap)
prices <- do.call(cbind, prices)
## now we have a matrix instead of a list
mcap <- mcap["1990/"] # subset to period from 1990
prices <- prices["1990/"]

## for each day, take only the first of the available mcaps/prices
## -> if there are more prices on one day, only the first is considered;
mcap <- apply.daily(mcap, FUN=function(x) x[1,] )
prices <- apply.daily(prices, FUN=function(x) x[1,] )
## to get the weights we need to divide the market cap of
## a single firm by the sum of all market caps
## since we want to weight time t returns with the market cap
## from t-1 we lag the caps by one time period
mcap_weights <- lag(mcap)/rowSums(lag(mcap), na.rm=TRUE)
## na.rm=TRUE removes all NA elements from a row prior
## to forming the rowsum
## we also need the returns
 r<- (prices/lag(prices)-1)


# remove first observation of 1990 (i.e. January),
# which is NA due to operating with lags
mcap_weights <- mcap_weights["1990-02/"]
r <- r["1990-02/"]

## to get the market return we value-weight all stock returns
r_mkt <- xts(rowSums(r*mcap_weights, na.rm=TRUE), order.by=time(r))
## Note: 'r' is the matrix of return time-series
## each column represents the return time-series of a firm
## 'r' has the same structure as 'mcap' but contains returns
## instead of the market capitalization
```

```
plot(r_mkt, col="steelblue", main="Market return")
```



4.2. Sorting stocks

A lot of investment strategies use stylized facts on stock returns to achieve return on investments that outperform some benchmark. For example, stocks that performed well over the past year tend to perform well in the near future and stocks that underperformed in the last year tend to underperform in the near future. This trading strategy is called *Momentum strategy*. Another example would be the *betting against beta strategy*. Essentially, you hold low beta stocks, i.e. stocks that contribute little to the market portfolio's volatility, and short sell high beta stocks. For an economic explanation see Frazzini and Pedersen (2013). For these and a lot of other strategies you need to rank the stocks/firms in the investable universe according to some variable, e.g. the momentum score, the beta, ... In this section we will have a look at how to do that in `R`.

4.2.1. `R` functions `sort`, `rank`, `order` and `cut`

There are several functions for sorting and ordering values. Let us take a look at the simplest, `sort`. `sort` takes the input vector and outputs the values of the input vector sorted increasingly. If you want the output the be decreasing, set `decreasing=TRUE`.

```
test <- c(5,3,4,7,6,2,1)
sort(test)

## [1] 1 2 3 4 5 6 7

sort(test, decreasing = TRUE)

## [1] 7 6 5 4 3 2 1
```

The function `rank` tells you the statistical or sample rank of the values in the input vector. This means the output vector tells you for each position in the input vector which rank it has. E.g. a rank of four means that it is the fourth lowest value.

```
test <- c(11,12,13)
rank(test)

## [1] 1 2 3

test <- c(13,12,11)
rank(test)

## [1] 3 2 1

test <- c(5,1,10,70,6,22)
rank(test)

## [1] 2 1 4 6 3 5
```

If you have ties, i.e. the same value twice, or `NAs` in you input vector, then you should specify how the function should handle them. See `?rank` for details on the different methods applicable.

The function `order` returns a vector of positions that indicates the order of the values in the input vector. So the output vector tells you which element of the input vector is the lowest, second lowest and so on. If you want the output vector to tell you the order decreasingly, set `decreasing=TRUE`.

```
test<-c(5,1,10,70,6,22)
order(test)

## [1] 2 1 5 3 6 4

order(test, decreasing=TRUE)

## [1] 4 6 3 5 1 2
```

You can use the output of `order` to sort the numbers of the input vector.

```
test<-c(5,1,10,70,6,22)
test[order(test)]

## [1]  1  5  6 10 22 70
```

In light of the function `sort` the `order` function looks rather useless since you can just apply `sort`, but this is not case. You can use `order` on one input vector, e.g. firm size and then use the output, i.e. the order of firms by size on another vector, e.g. their stock returns. To do this you only need to make sure that the positions in your firm size vector and stock return vector correspond to the same firm.

```r
##vector containing the size of all firms
firm.size

## Firm A Firm B Firm C Firm D Firm E
##    400    450    500    200    150

##vector containg the last monthly return of all firms
stock.return

## Firm A Firm B Firm C Firm D Firm E
##   0.04  -0.03  -0.02   0.10   0.08

##the order of all firms by size
order(firm.size)

## [1] 5 4 1 2 3

##the stock returns sorted by firm size
stock.return[order(firm.size)]

## Firm E Firm D Firm A Firm B Firm C
##   0.08   0.10   0.04  -0.03  -0.02
```

Finally, `cut` allows to sort the data by assigning each data point to a particular bin (interval). To define the boundaries of these bins, we can for example employ quantiles of the data. Bins are then defined as the inverals between two quantiles. In the example below, we devide the data into thirds (lower third, middle third, upper third).

```r
test <- 101:112
test <- sample(test, size=12, replace=FALSE) #randomly rearrange ordering
test

##  [1] 103 107 102 108 111 104 105 112 106 109 110 101

quantiles <- quantile(test, probs=c(0,1/3, 2/3, 1))
quantiles

##        0% 33.33333% 66.66667%      100%
##  101.0000  104.6667  108.3333  112.0000

bins <- cut(test, breaks=quantiles, include.lowest=TRUE)
# include.lowest=TRUE -> leftmost interval is left-closed
# instead of left-open
bins #factor with three levels corresponding to the thirds

##  [1] [101,105] (105,108] [101,105] (105,108] (108,112] [101,105] (105,108]
##  [8] (108,112] (105,108] (108,112] (108,112] [101,105]
## Levels: [101,105] (105,108] (108,112]
```

```r
# note: interval boundaries correspond to exact values in quantiles,
# but labels are indicated using rounded values
levels(bins) <- c("lower", "middle", "upper") #renaming bins
data.frame(test, bins)

##    test   bins
## 1   103  lower
## 2   107 middle
## 3   102  lower
## 4   108 middle
## 5   111  upper
## 6   104  lower
## 7   105 middle
## 8   112  upper
## 9   106 middle
## 10  109  upper
## 11  110  upper
## 12  101  lower
```

The object `bins` therefore indicates membership to the lower, middle or upper third of the data. We can now easily select all data assigned to a particular bin, or aggregate data within bins using `aggregate()`.

```r
# the bins need to be supplied as a list
aggregate(test, by=list(bins), FUN=mean)

##   Group.1     x
## 1   lower 102.5
## 2  middle 106.5
## 3   upper 110.5
```

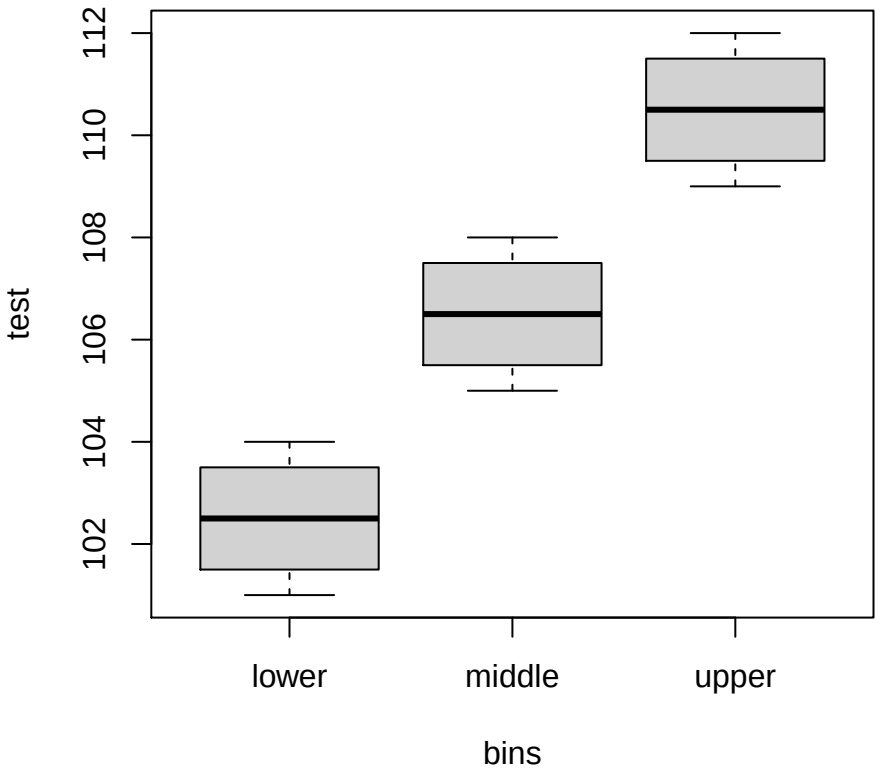
4.2.2. Sorting stocks repeatedly over time

Consider the following setup. You have data on monthly stock returns for a sample of investable firms and their corresponding market value of equity. Moreover, you have an idea for a simple trading strategy and you want to backtest your strategy on historical data before implementing it with your own money. The strategy works as follows.

1. At the beginning of every year you look at the average monthly market values of equity over the last year
2. Sort the stocks according to that average market value
3. Buy the thrity largest stocks (equally weighted) and hold them for the next year
4. After one year your repeat the steps

The dataset you work with has the following structure: A matrix containing monthly returns, where each column contains the time-series of monthly returns for a single firm,

```r
test[bins=="lower"]

## [1] 103 102 104 101

boxplot(test~bins)
```



and a matrix (`ex.sizes`) containing the market value of equity, where each column contains the time-series of market values (in monthly time increaments) for a single firm. So, each row contains the monthly corss-section of returns or market values. As to simplyfy matters a bit, we assume a return of zero for months in which we do not have a return observation.

```
## Warning:  package 'lubridate' was built under R version 4.0.3
```

```
library(xts)
library(lubridate)

## STEP ONE: compute the average size over each year
avg_sizes <- apply.yearly(ex.sizes,FUN=function(A){
  apply(A, 2, FUN=function(x) mean(x, na.rm=TRUE) )
})

## STEP TWO: order the firms by size
#could also use apply.yearly again
size_ordering <- rollapply(avg_sizes, width=1, by.column=FALSE,
                           FUN=function(x) order(x,decreasing=TRUE)
                           )

## As the stocks we buy THIS year depend on
## the average equity value (size) of LAST year,
## we need to lag the size ordering by one period
size_ordering <- lag(size_ordering) # first row now contains only NAs

## STEP THREE: pick only the largest 30 firms of the previous year
## and invest in them
## create a matrix where entry [t,i] is 1 if we want to hold firm i in year t,
## and zero otherwise
pf_stocks <- size_ordering
## if we dont hold the stock in particular year, set NA
pf_stocks[pf_stocks>30] <- NA
## set entries to 1 for all stocks we want to hold
pf_stocks[!is.na(pf_stocks)] <- 1
## convert NAs to 0 instead (easier to handle)
pf_stocks[is.na(pf_stocks)] <- 0
pf_stocks[1,] <- NA #restore NAs in first row, since we do not have
## information from previous year available

## next, convert the monthly returns to yearly
## (our strategy always holds stock over the full year)
## ASSUMPTION: if we do not have a price for a particular month,
## we assume a return of zero for this month
ex.returns[is.na(ex.returns)] <- 0
returns_yearly <- apply.yearly(ex.returns, FUN=function(A){
```

```
  apply(A,2,FUN=function(x) prod(1+x)-1)})

## finally, calculate the yearly portfolio return by forming
## the equally weighted return
## of only those stocks indicated in pf_stocks
## NOTE: for this to work, columns in the pf_stocks and returns_yearly
## must correspond to the same firms (which is the case here)
pf_ret <- apply.yearly(returns_yearly * pf_stocks,
                       FUN=function(x) sum(x)/30 )
# first year is NA, since there was no information on sizes
## in the year before available
```

```
## and plot it
par(mfrow=c(2,1))
plot(pf_ret, main="Large Firms Strategy")
ret_cum <- cumprod(1+pf_ret[-1]) #first year is NA
ret_cum <- c(pf_ret[1], ret_cum) #add back first date with NA
plot(ret_cum, main="Large Firms Strategy")
par(mfrow=c(1,1))
```

5. Hypothesis Tests

In this section the most important hypothesis tests and their corresponding R functions are introduced. We will cover nonparametric as well as parametric tests. The difference between these two types of tests is that *parametric tests* assume the data a test is used on to be drawn from some particular statistical distribution. *Nonparametric tests*, on the other hand, do not rely on such strong assumptions. Both types of tests have their advantages and disadvantages. Generally, nonparametric tests are more robust whereas parametric tests usually have more statistical power. [3] The lecture notes cover only the comparison of locations (i.e. sample averages), but note that there are also hypothesis tests comparing variances, correlations, trends in time-series, ... Also, please be aware that we will only state the most important assumptions – for a more detailed dive into hypothesis tests we refer to the courses Business Analytics I and Quantitative Methods.

5.1. Nonparametric Hypothesis Tests

The hypothesis test in this section include the Wilcoxon signed rank test and the Mann-Whitney-U test, also called the Wilcoxon rank-sum test.

We can use the Wilcoxon signed rank test compare two dependent, i.e. related samples or repeated measurements on the same sample. Moreover, the test can also be used to compare the sample location to a theoretical mean/location parameter – the one-sample Wilcoxon test.

Let us start with the one-sample Wilcoxon test. The null hypothesis, H_0, of the one-sample Wilcoxon test is

$$H_0 : X = x_0,$$

where X is the location statistic, in this case the median, and x_0 the theoretical, or *hypothesized* median. We consider one of three alternative hypotheses, H_A:

$$\begin{aligned} H_A : X < x_0 \quad & \text{left-tailed test,} \\ X > x_0 \quad & \text{right-tailed test,} \\ X \neq x_0 \quad & \text{two-sided test} \end{aligned}$$

This hypothesis structure is used in all hypothesis tests presented in the lecture notes.

Before showing you how to use R to do a one-sample Wilcoxon test it is important to be aware of the assumptions this test makes.

1. The observations in the sample are mutually independent.
2. The observations are (theoretically) continuous.
3. The data follows a symmetric distribution.
4. The data are measured at least on an interval scale.

[3] A robust statistic is a statistic that performs well regardless of the underlying distribution of the data and the presence of outliers. Statistical power refers to the probability of rejecting the null hypothesis given that the alternative hypothesis is true, i.e. the probability of avoiding a so-called typoe II error.

R provides us with the function `wilcox.test()`. This funciton is a multi purpose function as it can be used for the one- and two-sample Wilcoxon signed-rank test and the two-sample Wilcoxon rank-sum test.[4] When we want to do a one-sample test then we need to set the parameter `x` equal to the sample we want to test and leave `y=NULL`, which is the default. Moreover, we need to tell R the theoretical median via the argument `mu` and what kind of alternative hypothesis we want to test. If we want to do a two-sided test, then we need to set `alternative="two.sided"`. For a left-tailed test set `alternative="less"` and for a right-tailed test set `alternative="greater"`.

```
## draw a sample of 10000 random numbers
## that are uniformly distributed between 1 and 100
test_data <- runif(n = 10000, min = 1, max = 100)
## the theoretical median (and the theoretical mean)
## of a uniform distribution
## are the center of the interval, i.e. the center of [1,100]
## => (100+1)/2 = 50.5
x0 <- 50.5
wilcox.test(x=test_data, mu=x0, alternative = "two.sided")

##
##  Wilcoxon signed rank test with continuity correction
##
## data:  test_data
## V = 25002427, p-value = 0.9998
## alternative hypothesis: true location is not equal to 50.5
```

Usually, we only check the p-value when looking at the output from `wilcox.test()`, as the test statistic itself is rather uniformative because it can only be interpreted when one adjusts it for the number of observations. Unsurprisingly, we do not reject H_0!

Next, we take a look at the two-sample Wilcoxon signed-rank test which is used for dependent samples – so either if we consider two observations of the same entities over time or two related observations. The alternative hypotheses to consider are the same as above and the assumption of this test in principle also coincide with the ones above although on a pair's level. Consider two samples

```
test_data1 <- runif(n = 10000, min = 1, max = 100)
## simulate an increase over time
test_data2 <- test_data1+rnorm(n = 10000, mean = 1, sd = 0.5)
```

Note that `test_data1` and `test_data2` are dependent, since `test_data2` was created adding some random normally distributed quantity (with mean 1 and standard deviation 0.5) to the observations of `test_data1` – if e.g. the first observation in `test_data1` is particularly large (close to 100), also the corresponding paired observation in `test_data2` will tend to be rather large. Obviously, the median of the first sample should be below the median of the second sample, since the normally distributed term that we added to `test_data2` has a mean of 1. To check this claim we use `wilcox.test()`. Since we want to compare two dependent samples we need to set `paired=TRUE`.

[4]The two-sample Wilcoxon rank-sum test is also known as Mann-Whitney U Test.

```
wilcox.test(x = test_data1, y = test_data2, alternative = "less",
            paired = TRUE)

##
##  Wilcoxon signed rank test with continuity correction
##
## data:  test_data1 and test_data2
## V = 124525, p-value < 2.2e-16
## alternative hypothesis: true location shift is less than 0
```

The highly significant p-value leads us to reject H_0 in favor of $H_A : X_{t+1} > X_t$.

Lastly, we take a look at the Mann-Whitney U test or Wilcoxon rank-sum test which is used to compare two independent samples. The assumptions underlying this test are

1. The observations in the sample are mutually independent.
2. The data are measured at least on an ordinal scale.

Here the null hypothesis is that the population distributions are equal and the alternative is that the population distributions are not equal (two-sided test) or that one of the distributions is (stochastically) greater [smaller] than the other (righ-tailed [left-tailed] test). We can use the Mann-Whitney U test to check whether the returns of two groups of stocks come from the same distribution or not. In the following example the returns of firms from January 1995 are split into 5% quantile-bins according to size. We want to answer the question whether the returns of the smallest 5% of firms (`r_small`) have the same distribution as the returns of the largest 5% of firms (`r_large`) using the Mann-Whitney U test.

```
wilcox.test(x = r_small, y = r_large, alternative = "two.sided")

##
##  Wilcoxon rank sum exact test
##
## data:  r_small and r_large
## W = 189, p-value = 0.01607
## alternative hypothesis: true location shift is not equal to 0
```

According to the Mann-Whitney U test, the two groups have a different return distribution.

5.2. Parametric Hypothesis Tests

Parametric hypothesis tests are hypothesis test that assume that the underlying data are drawn from a particular population distribution. In the lecture notes we will show you how to carry out a one- and two-sample t-test — the most common hypothesis test. The t-test, sometimes the Student's t-test, is named after the test statistic one computes, i.e. the t statistic. Generally, we have that $t := F/G$, where F, and G are functions of the data.[5] In the following we will look at the one- and two-sample location t-test.

[5] Student was the pseudonym of the statistician who invented the t-test. Fun fact, he developed it for the Guiness Brewery, the company he was working for.

In case of the one-sample location test, the t statistic is defined as $t := \frac{\bar{x}-\mu}{s/\sqrt{n}}$, where $\bar{x}$ is the sample mean, s the sample standard deviation, n the number of observations and μ the theoretical mean you want to test. The assumptions for the one-sample t-test are

1. $\bar{x} \sim \mathcal{N}\left(\mu, \frac{\sigma^2}{n}\right)$
2. $\frac{s^2(n-1)}{\sigma^2} \sim \chi^2\,(n-1)$
3. F and G are independent.
4. The data are continuous.
5. The sample is a random sample from the population.

If the underlying data are i.i.d. normal, then assumptions one and two are met. However, even if the underlying data are not normal and we we have a sufficiently large sample size, e.g. $n > 30$, assumption one is met by the central limit theorem. Moreover, under these conditions Slutky's theorem states that the distribution of the sample variance does not greatly influence the distribution of the test statistic, i.e. the t statistic.

`R` provides us with the function `t.test()` which we can again use for one- and two-sample tests. The parameter `x` contains the data, `mu` the theoretical mean we want to test and `alternative` the alternative hypothesis. We can use a one-sample t-test to check if the January 2015 returns (`r_20151`) are on average zero.

```
t.test(x = r_20151, mu = 0, alternative = "two.sided")

##
##  One Sample t-test
##
## data:  r_20151
## t = -6.8465, df = 383, p-value = 3.021e-11
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  -0.05045452 -0.02794093
## sample estimates:
##   mean of x
## -0.03919773
```

Given a very low p-value we can reject the null hypothesis that the January 2015 returns are on average zero. Note that `R` also outputs the value of the test statistic and the degrees of freedom. More on this later.

The two-sample t-test can again be done for dependent and independent samples. The general assumptions for the two sample t-test are

1. The means of the populations follow a normal distribution.
2. The variances of the populations are equal.
3. The data are continuos.
4. The observations should be independent or fully matched, i.e. paired.

Assumption one is again satisfied by the central limit theorem for relatively large sample sizes. Student's t-test is relatively robust to deviations from assumption two, but Welch's t-test does not need this assumption at all.

Using the argument `paired` we can tell `R` if we want to conduct a dependent (`paired=TRUE`) or independent (`paired=FALSE`) t-test. By default, `var.equal=FALSE` and `R` conducts a Welch t-test, as the assumption of equal variances is more often than not violated. However, if you know that the population variances are the same, then set `var.equal=TRUE` and `R` will use the pooled sample variances to estimate the population variance.

We can use an independent t-test to again compare the returns from small firms to the returns of big firms in January 1995.

```
t.test(x = r_small, y = r_large, alternative = "two.sided")

##
##  Welch Two Sample t-test
##
## data:  r_small and r_large
## t = -0.81082, df = 27.478, p-value = 0.4244
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.13852559  0.06000781
## sample estimates:
##   mean of x   mean of y
## 0.009459381 0.048718269
```

According to the t-test the means are not different. Remember, when conducting a Mann-Whitney U test, we concluded that the two distributions are unqueal. So what is going on? Let us take a brief look at the six point summaries of the two distributions

```
summary(r_small)

##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.      NA's
## -0.250888 -0.117292 -0.000113  0.009459  0.042586  0.971449         4

summary(r_large)

##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.     NA's
## -0.15749  0.02826  0.05939  0.04872  0.09555  0.14330        4
```

According to these summaries the two distribution are not the same! But why are the means not statistically different? The answer is that the 95% confidence intervals overlap. The bootstrapped 95% confidence interval for the mean of the small firm returns is $[-0.062, 0.088]$ and for the mean of the large firms $[0.027, 0.068]$. The confidence interval of the small firms completely contains the confidence interval of the large firms and thus the means are not different. The Mann-Whitney U test, on the other hand, considers the medians. The bootstrapped 95% confidence intervals of the medians is for the small firms $[-0.036, 0.033]$ and for the large firms $[0.035, 0.078]$ and do not overlap. If we would compute the 99% confidence intervals, then also the median confidence intervals would overlap. Please be aware that this is still in line with out results. Using a confidence level of 99% we would not reject the null hypothesis in the Mann-Whitney U test!

A dependent t-test can be used to compare the returns from January 2015 and February 2015, we just have to make sure that for each data point (i.e. individual firms' returns) we have an observation in January and Febraury.

```
## note that the data is structured in a way
## that r_20151 and r_20152 are paired!
t.test(x = r_20151, y = r_20152, paired = TRUE, var.equal = FALSE)

##
##  Paired t-test
##
## data:  r_20151 and r_20152
## t = -3.376, df = 382, p-value = 0.0008109
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.19444855 -0.05131505
## sample estimates:
## mean of the differences
##              -0.1228818

## we use this example to demonstrate that the Student's t-test
## is very robust to unequal variances
t.test(x = r_20151, y = r_20152, paired = TRUE, var.equal = TRUE)

##
##  Paired t-test
##
## data:  r_20151 and r_20152
## t = -3.376, df = 382, p-value = 0.0008109
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.19444855 -0.05131505
## sample estimates:
## mean of the differences
##              -0.1228818
```

The return distribution does not seem to be stable over time. Note that we conducted a Welch's t-test and a Student's t-test to demonstrate that the Student's t-test is very robust to unqueal variances (especially when using large samples).

6. Measuring Risk

In this section, we introduce several measures commonly used to quantify the risk inherent in the evolution of returns of individual stocks or investment portfolios. Throughout the section, we are going to consider simple returns. For now, furthermore only consider the first firm in our sample.

```
r1 <- simple.r[,1] # subset out data to only the first column,
# i.e. first firm
head(r1)

##                    X10001
## 1990-02-28 -0.0079257711
## 1990-03-30  0.0008407991
## 1990-04-30  0.0038604096
## 1990-05-31 -0.0172948928
## 1990-06-29  0.0048971297
## 1990-07-31  0.0215810735
```

6.1. Volatility

We can compute the *variance* of all elements contained in a vector using the function `var()`. To arrive at the *standard deviation*, or *volatility*, we need to take the square root of the variance. Equivalently, however, the volatility (synonymous to standard deviation) can also be directly obtained using `sd()`.

```
var(r1)

##             X10001
## X10001 0.006550258

sqrt(var(r1))

##            X10001
## X10001 0.08093366

sd(r1) # equal to sqrt(var(r1))

## [1] 0.08093366
```

Under the assumption that each month's return is uncorrelated with the previous month's return, the monthly volatility obtained from our monthly time series can easily be converted to yearly[6]:

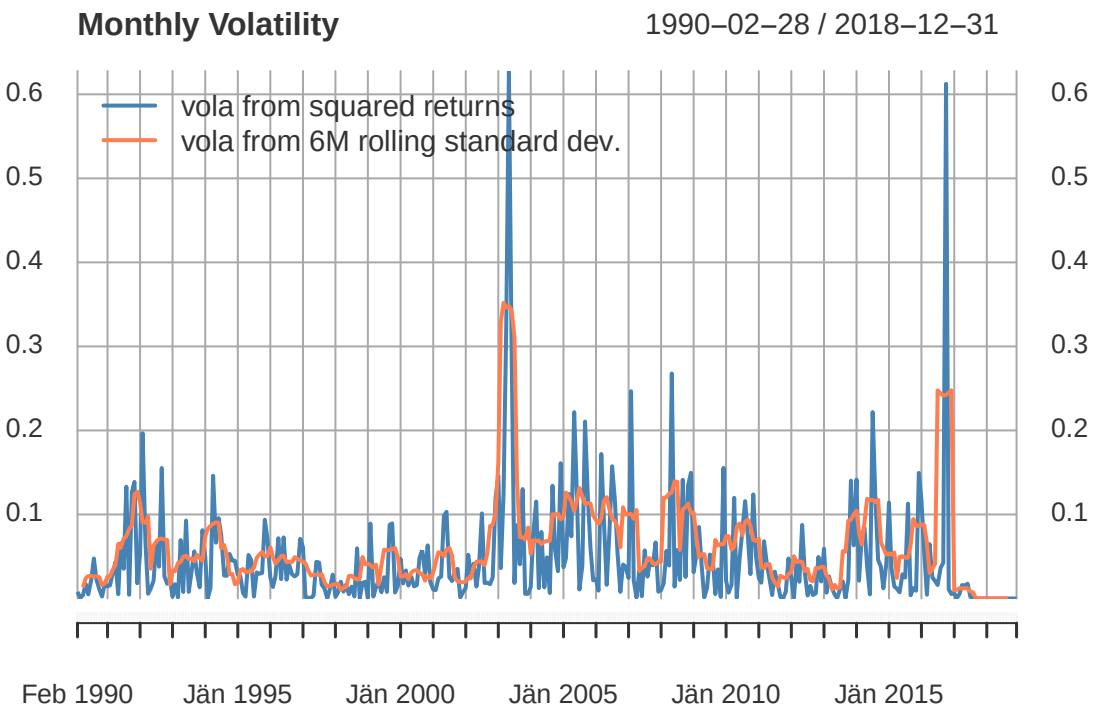
```
var(r1)*12 # yearly varaince over whole sample period

##           X10001
## X10001 0.0786031
```

[6] This approach is not precise, but common practice.

```r
sd(r1)*sqrt(12) # yearly volatility over whole sample period

## [1] 0.2803624
```

If we are not interested in volatility over the whole sample period, but rather want to learn about the evolution of volatility over time, two popular options include:

```r
# two measures of monthly volatility:
# square root of squared returns
# (you cold also take absolute value right away)
# and variance over a rolling window
plot(sqrt(r1^2), col="steelblue", lwd=2, main="Monthly Volatility")
lines(rollapply(r1, width=6, align="center", FUN=sd), lwd=2, col="coral")
#window of 6 months
addLegend("topleft", legend.names=c("vola from squared returns",
                                    "vola from 6M rolling standard dev."),
          col=c("steelblue", "coral"), lwd=2)
```



Note, however, that in general, we would prefer to assess monthly volatility levels by aggregating-up data from a higher frequency (e.g. daily data). Starting from monthly data, we estimate volatility only with very few data points.

6.2. Covariance and correlation

To compute the covariance of multiple timeseries, use `cov()`. However, you can equivalently also use `var()` as in the univariate case. Correlations can be computed using `cor()`.

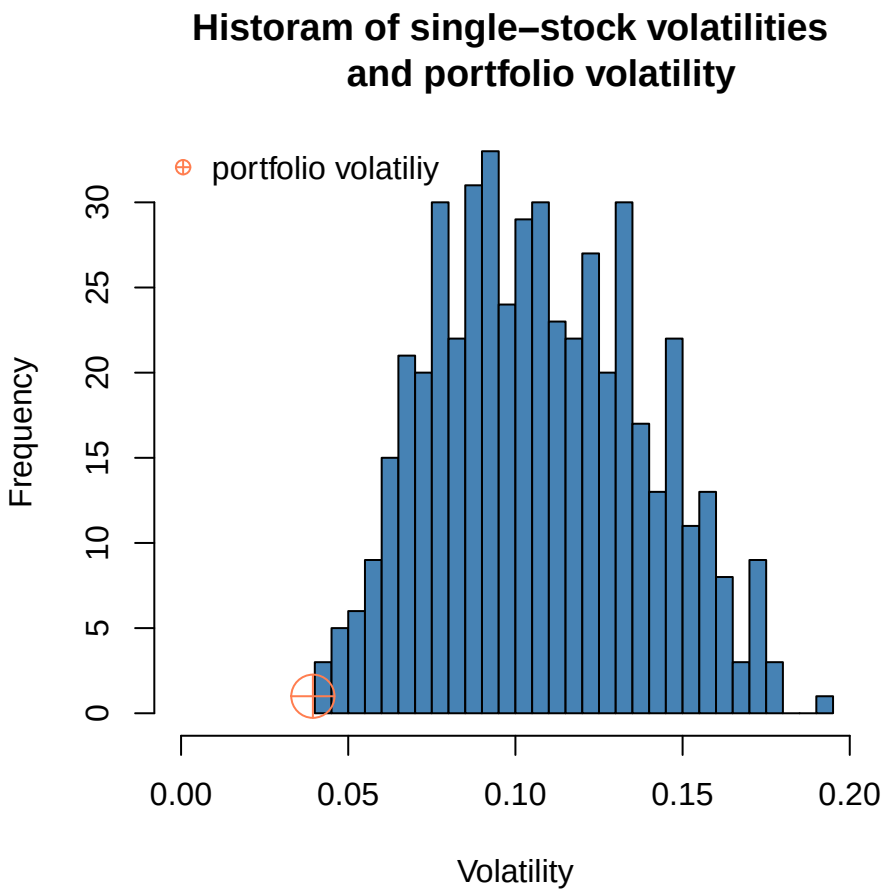
```r
# subsetting our data to the first 5 companies
r1_5 <- simple.r[,1:5]

# covariance
round(var(r1_5), digits=4) #round to display just a few digits

##         X10001  X10002 X10025 X10026 X10028
## X10001  0.0066 -0.0002 0.0002 0.0000 0.0024
## X10002 -0.0002  0.0100 0.0007 0.0012 0.0016
## X10025  0.0002  0.0007 0.0198 0.0029 0.0003
## X10026  0.0000  0.0012 0.0029 0.0080 0.0009
## X10028  0.0024  0.0016 0.0003 0.0009 0.0412

round(cov(r1_5), digits=4) #identical to var()

##         X10001  X10002 X10025 X10026 X10028
## X10001  0.0066 -0.0002 0.0002 0.0000 0.0024
## X10002 -0.0002  0.0100 0.0007 0.0012 0.0016
## X10025  0.0002  0.0007 0.0198 0.0029 0.0003
## X10026  0.0000  0.0012 0.0029 0.0080 0.0009
## X10028  0.0024  0.0016 0.0003 0.0009 0.0412

# correlation
round(cor(r1_5), digits=4)

##         X10001  X10002 X10025  X10026 X10028
## X10001  1.0000 -0.0203 0.0188 -0.0049 0.1490
## X10002 -0.0203  1.0000 0.0491  0.1389 0.0782
## X10025  0.0188  0.0491 1.0000  0.2333 0.0091
## X10026 -0.0049  0.1389 0.2333  1.0000 0.0475
## X10028  0.1490  0.0782 0.0091  0.0475 1.0000
```

Finally, let's consider a simple portfolio that invests an equal amount in every stock available in our portfolio (the weights are readjusted every day to keep that balance), and compare the volatility of this portfolio to the volatilities of all individual stocks.

```r
# some cleaning of the raw returns data
# boundaries are most extreme 1% of returns
quantile(simple.r, probs=c(0.005, 0.995), na.rm=TRUE)

##       0.5%      99.5%
## -0.3709735  0.5876805

qlower <- quantile(simple.r, probs=0.005, na.rm=TRUE)
qupper <- quantile(simple.r, probs=0.995, na.rm=TRUE)

dates <- time(simple.r)
simple.r <- coredata(simple.r) #stirp away xts properties
simple.r[simple.r<qlower] <- NA #remove extreme values to the left
```

```r
simple.r[simple.r>qupper] <- NA #remove extreme values to the right
simple.r <- xts(simple.r, order.by=dates) #convert back to xts

# equally weighted portfolio
r_pf <- rowMeans(simple.r, na.rm=TRUE)
r_pf <- xts(r_pf, order.by=time(simple.r))


vola_single <- apply(coredata(simple.r), 2,
                     FUN=function(x) sd(x, na.rm=TRUE) )
vola_pf <- sd(r_pf, na.rm=TRUE)

summary(vola_single)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.04183 0.08409 0.10555 0.10733 0.13140 0.19339

vola_pf

## [1] 0.03942599
```

```r
hist(vola_single, breaks=50, xlim=c(0,0.2),
     xlab="Volatility",
     main="Historam of single-stock volatilities
     and portfolio volatility",
     col="steelblue")
points(x=vola_pf,y=1, col="coral", pch=10, cex=3)
# bty="n" -> no box is drawn around the legend
legend("topleft", legend="portfolio volatiliy", pch=10, col="coral", bty="n")
```



6.3. Skewness

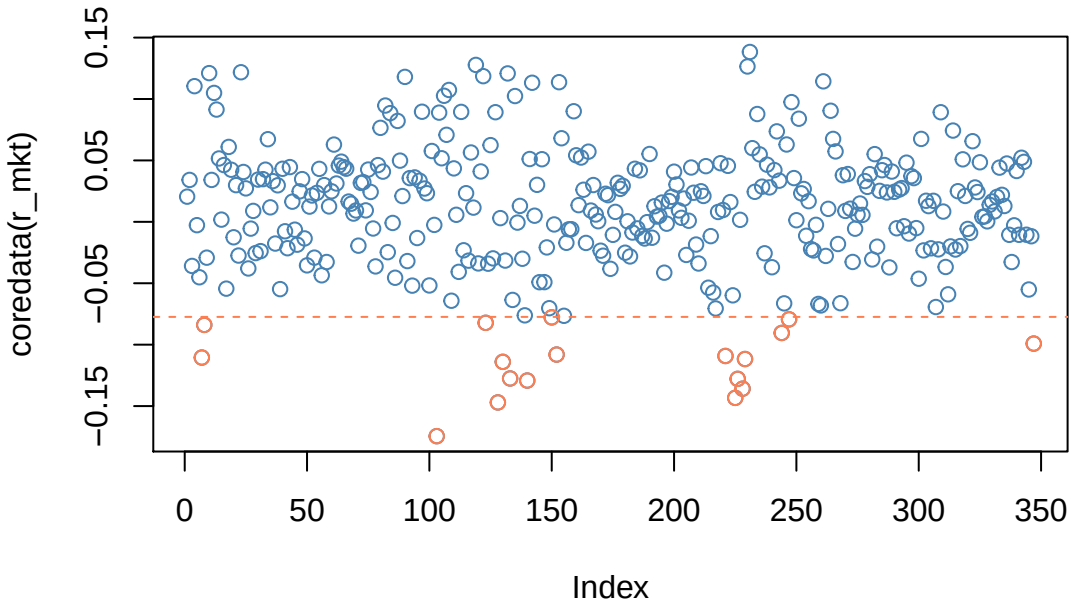
A function for skewness is not provided in `R` by default, but can be loaded using the `moments` package – the relevant function there is called `skewness()`. In the following, let's consider the return of the market computed at the end of section 4.1.

```r
library(moments)

## Warning:  package 'moments' was built under R version 4.0.3

skewness(r_mkt, na.rm=TRUE)

##           X
## -0.3820282
```

6.4. Value at Risk and Expected Shortfall

By the definition of the Value at Risk (VaR), we are looking for a particular quantile of the distribution of returns. Therefore, we can compute the VaR as:

```
r_mkt <- na.omit(r_mkt)

VaR95 <- quantile(r_mkt, probs=0.05)
VaR95

##          5%
## -0.07739907

plot(coredata(r_mkt), col="steelblue")
abline(h=VaR95, col="coral", lty=2)
points(x=which(coredata(r_mkt)<VaR95),
       y=coredata(r_mkt)[coredata(r_mkt<VaR95)], col="coral" )
```




```
# extreme losses are not independent over time!
```

Once we have obtained the VaR, we can also consider the average of only those losses that exceed the VaR – the Expected Shortfall.

```
mean(r_mkt[r_mkt < VaR95])

## [1] -0.1139754
```

6.5. Sharpe Ratio

To compute the sharpe ratio, we need a risk-free rate of return. Here, we consider the risk-free rate provided by Kenneth R. French on his website, included in the last column of `ff_rf.CSV`.

```
rf <- read.csv("../data/risk_free_return_monthly.csv")
head(rf)

##     date   rf
## 1 192607 0.22
## 2 192608 0.25
## 3 192609 0.23
## 4 192610 0.32
## 5 192611 0.31
## 6 192612 0.28

rf$date <- as.Date(paste0(rf$date, "01"), format="%Y%m%d")
rf<-xts(rf$rf, order.by = rf$date)

rf <- rf/100 #data is in percent
rf <- rf["1990-02/2018-12"] #extract the desired time-frame

#check that we have indeed the same time points for r_mkt and rf
all(as.character(time(r_mkt))==as.character(time(rf)))

## [1] FALSE

# round date to 1st of next month and move one day back
time(r_mkt) <- ceiling_date(time(r_mkt), unit="month") -1
time(rf) <- ceiling_date(time(rf), unit="month") -1

all(as.character(time(r_mkt))==as.character(time(rf)))

## [1] TRUE

head(cbind(r_mkt, r_mkt-rf))

##                    r_mkt   r_mkt...rf
## 1990-02-28  0.020505491  0.014805491
## 1990-03-31  0.034124502  0.027724502
## 1990-04-30 -0.035916566 -0.042816566
## 1990-05-31  0.110433108  0.103633108
## 1990-06-30 -0.002724144 -0.009024144
## 1990-07-31 -0.045141393 -0.051941393

rex_avg <- mean(r_mkt-rf) # average monthly excess return

#annualized sharpe ratio
((1+rex_avg)^12 - 1) / (sd(r_mkt-rf)*sqrt(12))
```

```
## [1] 0.5290847

# annualized SR using geo. mean
( prod(1+r_mkt-rf)^(12/length(r_mkt)) -1) / (sd(r_mkt-rf)*sqrt(12))

## [1] 0.4321974
```

7. Numerical Optimization

The no-short-selling problem of a portfolio manager.

In this section we will take a look at the numerical optimization capabilities of `R`. To illustrate the task we consider the optimization problem of a portfolio manager. The portfolio manager is a mean-variance optimizer and has to construct a portfolio without any short selling. Formally, the portfolio manager has to solve

$$\begin{aligned} \min_w \frac{1}{2} w'\Sigma w \quad \text{subject to } & \mu' w = \mu_P, \\ & \mathbf{1}' w = 1, \\ & w \geq 0. \end{aligned} \tag{2}$$

The portfolio variance is $w'\Sigma w$, where

- w is the vector of portfolio weights that should be chosen optimally.
- Σ is the portfolio covariance matrix (i.e. covariance matrix of individual stocks making up the portfolio).
- μ is the vector of expected returns (of individual stocks).
- μ_P is the expected return of the portfolio.
- $\mathbf{1}$ is a vector containing only ones.

This problem has three conditions: First, $\mu' w = \mu_P$ states that the weighted sum of expected returns μ has to be equal to the expected portfolio return μ_P. Second, $\mathbf{1}' w = 1$ states that the portfolio weights need to sum up to one and third, $w \geq 0$ is the no short selling constraint, i.e. all weights have to be non-negative.

To solve this problem with the help of `R` we will use the function `solve.QP` from the package `quadprog`. `?solve.QP` tells you that `solve.QP` solves problems of the form

$$\min -d'b + \frac{1}{2} b' D b \quad \text{subject to } A' b \geq b_0. \tag{3}$$

This is actually very similar to (2), we just need to set $d = 0$, $b = w$ and $D = \Sigma$. The major difference to the portfolio manager's problem is how the constraints are stated. We started by stating every of our three constraints explicitly. The `quadprog` pakcage, however, states the constraints with the help of constraint matrix A and constraint vector b_0. We need to figure out how to translate the constraint formulation from the portfolio manager's problem to the one of the `quadprog` package. Obviously, b_0 represents the right hand side of the constraints. The right hand side of the first two constraints is a number, so the first entry of b_0 has to be the expected portfolio return μ_P and the second entry has to be a one. The right hand side of the third constraint however, is a vector of zeros. More specifically, of as many zeros as there are weights and thus investable assets. Therefore, the dimension, $\bar{n}$, of b_0 is the number of investiable assets (n) plus two. The left hand side of the constraints inequality is $A'b$, and since the right hand side is of dimension $\bar{n} \times 1$ also the left hand side has to be of dimension $\bar{n} \times 1$. We already determined that b is of dimension n, because $b = w$ and thus, A has to be of dimension

$n \times \bar{n}$ and equivalently A' of dimension $\bar{n} \times n$. Let us look at $A'b \geq b_0$ in more detail. When multiplying the first row of A' with b the result is compared to the first element of b_0, i.e. μ_P. Hence, the first row of A' times b should be the same as $\mu' w$ and since b and the first row of A' has n entries, we can set the first row of A' equal to μ'. The second row of A' times b is compared the second entry of b_0, which is one. So the second row of A' models the second constraint. Comparing this to $\mathbf{1}'w$ we see that the second row of A' has to be full of ones. Finally, the third constraint. We want that every element of w is non-negative and thus every element of $b \geq 0$. A' has n colums and $\bar{n}$ rows and the first two rows are already used, so we are left with n rows, i.e. a $n \times n$ matrix. Can you figure out how this matrix needs to look like when we want the output of $A'b$ (without the first two rows) to be w? Exactly, the remaining $n \times n$ matrix needs to be the identity matrix. If we consider an example with only three assets, the constraints look like this, with the inequalities for the first two elements of the right hand side vector (μ_P and 1) actually binding (i.e. equality constraints):

$$\underbrace{\begin{bmatrix} \mu_1 & \mu_2 & \mu_3 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{A'} \cdot \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}}_{b} \geq \underbrace{\begin{bmatrix} \mu_P \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{b_0} \tag{4}$$

Maybe you have noticed that the first constraints in (2) are equality constraint and the corresponding constraints in (3) are inequality constraints. Luckily, we can tell `solve.QP` via the argument `meq` how many of the constraints used are equality constraints. The only thing we need to make sure is that the equality constraints are modeled first, as `solve.QP` will treat the first `meq` constraints as equality constraints and the remaining constraints as inequality constraints.

Before we start with the code, let us briefly think about the inputs μ and Σ. μ is the vector of **expected** asset returns. The question which numbers to put into μ is, as discussed in class, a very difficult one to answer. However, this course is mostly concerned with teaching you how to use `R` for finance purposes and not with finance theory. Thus, as a proxy for expected future stock returns, we will just use the historical average of realized stock returns for μ. The same holds for Σ, i.e. we will compute Σ as the historical covariance matrix. At the end of this chapter we talk about possible problems when estimating Σ.

In the following example we use the returns obtain from the price time-series we provided to you.

```
## The returns are stored in an object called 'r'
dim(r)

## [1] 346 127

head(r[,1:5])

##                 X10026      X10107      X10138       X10308       X10375
## 1990-03-30 -0.03329589 -0.01675624  0.05307179 -0.007914564 -0.068404127
## 1990-04-30  0.04949759  0.15845686 -0.19554238 -0.007117822 -0.211581686
```

```
## 1990-05-31 -0.03719380  0.13497300  0.11787967 -0.025213755  0.002045399
## 1990-06-29 -0.06038468  0.07882447  0.09650792 -0.045355140 -0.002363582
## 1990-07-31  0.03643682 -0.11453219 -0.11578095 -0.085514509 -0.079881356
## 1990-08-31 -0.09677185 -0.01067557 -0.19177697  0.031895819 -0.074833552

summary(time(r))

##         Min.      1st Qu.       Median         Mean      3rd Qu.         Max.
## "1990-03-30" "1997-06-06" "2004-08-15" "2004-08-14" "2011-10-23" "2018-12-31"
```

Now that you know what data we are dealing with, let us define d, A' and b_0.

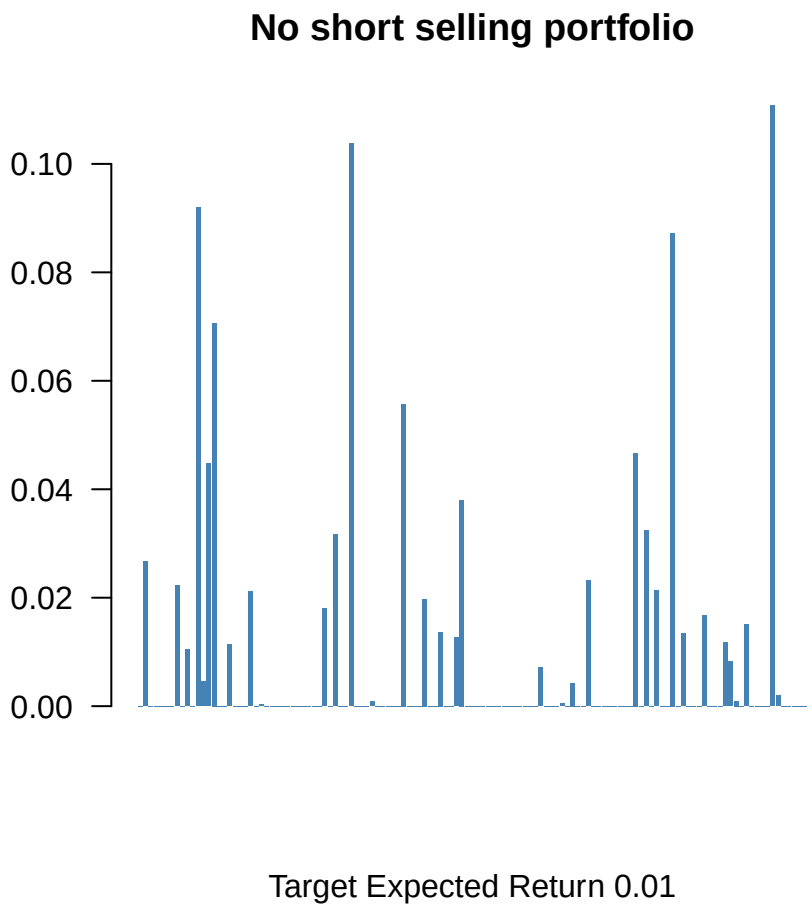
```
library(quadprog)

## Warning:  package 'quadprog' was built under R version 4.0.3

n <- ncol(r) # get the number of assets
## we dont want the first part of the solve.QP problem, i.e. -d'*b
## to enter into the optimization
## and thus, set d=0
dvec <- rep(0, n)

## create a vector of expected stock returns
## use the historical average return as proxy
mu <- colMeans(r, na.rm=TRUE)

## compute the variance-covariance matrix of stock returns
S <- cov(r)

## Code the constraints

A_1 <- mu # weighted expected returns need to be expected PF return
A_2 <- rep(1,n) # weights need to sum to one
A_3 <- diag(x = n) # Short selling constraint

A <- t( rbind(A_1,A_2,A_3) )
## this matrix represents the LHS of all constraints
```

Since we want to get the portfolio frontier we need to compute the minimum variance portfolios for a range of expected portfolio returns. Before we do that, we show you how the optimization works for a single expected portfolio return.

```
mu.pf <- 0.01 #set the target expected portfolio return
b0 <- c(mu.pf, 1, rep(0,n)) #this vector the RHS of constraints

w.solution <- solve.QP(Dmat=S, dvec=dvec, Amat=A, bvec=b0, meq=2)
## don't forget to set meq=2 to indicate
## that the first two rows contain equality constraints
## solve.QP looks for portfolio weights that minimize
```

```
## the portfolio variance  for a target return of 0.01


head(w.solution$solution) #weights of the minimum variance portfolio

## [1]  2.492712e-17  2.666602e-02 -1.423728e-16  7.936810e-19 -3.499495e-17
## [6]  6.300503e-19

w.solution$value*2 #this is the variance of that portfolio

## [1] 0.000606619

## (since solve.QP minimizes 1/2 times the portfolio variance)
```

```
## we can use a barplot to visualize all portfolio weights.
## No short selling. To get 0.01 expected return it is sufficient to invest
## in a fraction of assets.
barplot(w.solution$solution, col = "steelblue", border = FALSE, las=2,
            main="No short selling portfolio",
        sub="Target Expected Return 0.01")
```

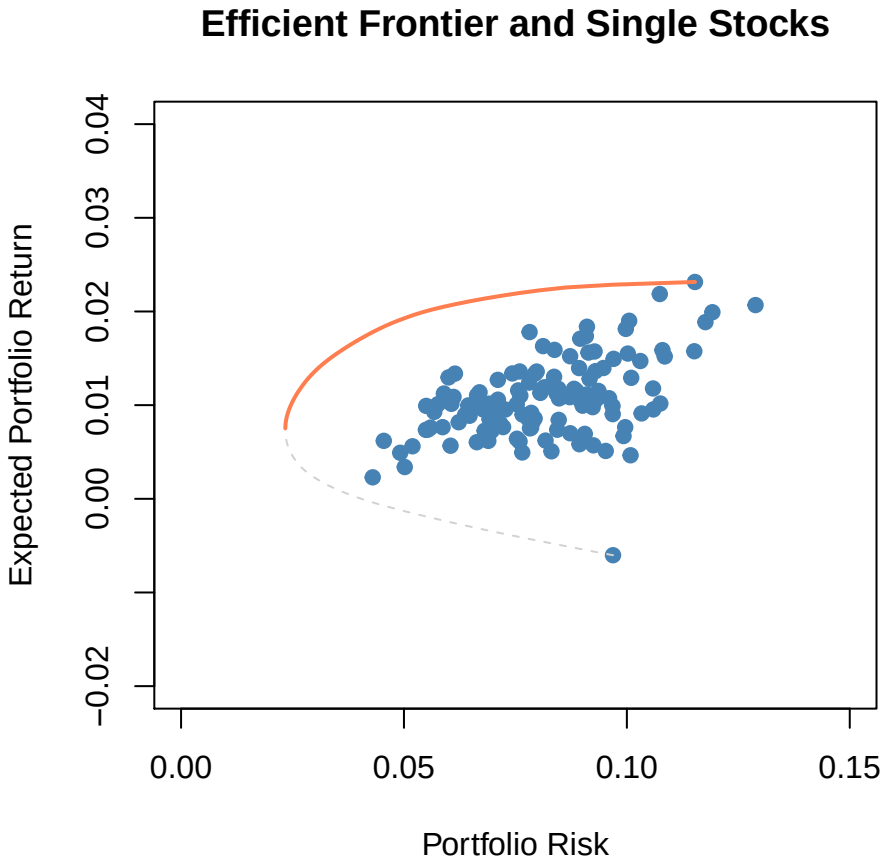


Now that we know how `solve.QP` works, we can compute the portfolio frontier for a

range of expected portfolio returns.

```
## To find the no-short-selling frontier we need solve the above problem
## for a range of expected portfolio returns!

## set the range of expected portfolio returns
frontier.r <- seq(min(mu), max(mu), length.out=100)

## create a list to store the output of solve.QP
## each list element will contain the output for one particular value of
## the expected portfolio return
## (all target values are contained in frontier.r)
w.solution <- list()
for (i in 1:length(frontier.r)) {
  b0 <- c(frontier.r[i], 1, rep(0,n)) #RHS of constraints
  w.solution[[i]] <- solve.QP(Dmat=S, dvec=dvec, Amat=A, bvec=b0, meq=2)
}



## go to each list-element extract the volatility
## from the contained solution
frontier.s <- rep(NA, length(w.solution))
for(i in 1:length(frontier.s)) {
  frontier.s[i] <- sqrt(w.solution[[i]]$value*2)
}

## NOTE: a more elegant implementation of this very last for-loop would be
frontier.s_version2 <- sapply(X = w.solution,
                              FUN = function(x) sqrt(x$value*2))
```

```r
## Plot the stocks in the risk-return plane,
## the portfolio and the efficient frontier
min_var_pf <- which.min(frontier.s)
plot(sqrt(diag(S)),mu,pch=19, col="steelblue", xlim = c(0,.15),
     ylab = "Expected Portfolio Return", xlab="Portfolio Risk",
     main="Efficient Frontier and Single Stocks", ylim = c(-.02,.04))
lines(frontier.s,frontier.r, lty=2, col="lightgrey")
lines(frontier.s[min_var_pf:length(frontier.s)],
      frontier.r[min_var_pf:length(frontier.r)], col="coral",lwd=2)
```



We found all minimum variance portfolios for expected protfolio returns in the range $[-0.006, 0.023]$ with the help of `solve.QP`. The portfolio frontier is far left of the single stocks indicating that a diversified portfolio of stocks decreases risk drastically while delivering the same expected return.

```r
## sapply applies the function defined via FUN to every list element
## of w.solution and the results are combined into a vector
## we need to check for >0.00001 and not >0 due to
## numerical imprecision of the solution
no_of_stocks_in_pf <- sapply(w.solution,
                             FUN = function(x) sum(x$solution>0.00001))
summary(no_of_stocks_in_pf)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    4.00   16.50   16.63   27.50   34.00
```

On average 17 out of 127 stocks are in the minimum variance portfolio.[7] Moreover, the average volatility of the efficient portfolios is 0.06 compared to the average volatility of stocks (0.08) this is a decrease of 25% that can be achieved by diversification. In contrast, the average expected return of all efficient portfolios is 0.015 compared to 0.010 average expected single stock returns.

Possible problems when estimating Σ.

The sample covariance matrix of an observed sample of T independent observations of a n-dimensional vector x is given by

$$\bar{\Sigma} = \frac{1}{T-1} \sum_{t=0}^{T} (x_t - \bar{x}) (x_t - \bar{x})', \tag{5}$$
$$\text{where } \bar{x} = \frac{1}{T} \sum_{t=0}^{T} x_t.$$

Assume that x_t are realizations of the random varialbe X. We only need that the mean and the covariance of X exist, the distribution of X does not matter for the sample covariance. However, there is a problem. This sample covariance matrix is not guaranteed to be positive definite, although in theory a covariance matrix is. Why is this a problem? Consider the equation for the optimal portfolio weights from the slides

$$w^* = \left((\lambda_1 \mu' + \lambda_2 \mathbf{1}') \, \Sigma^{-1} \right)'. \tag{6}$$

In (6) the problem becomes apparent. In theory, Σ is always invertible because it is guaranteed to be positive definite, but $\bar{\Sigma}$ is not guranteed to be positive definite. Usually, this is because $\bar{\Sigma}$ does not have full rank or there are some missing data points in x_t. If you get an error in `R` that reads something like *system is computationally singular*, then this usually means that some of your observations are highly correlated or maybe even linearly dependent.

Another problem with the estimation of Σ, is that for large n and large t, the compuation becomes exmtremly cumbersome and you may need to resort to other methods for computing $\bar{\Sigma}$. We won't go into details here, but refer the interested reader to Coqueret and Milhau (2014).

[7] There are two portfolios cosisting only of one stock, these are actually the *boundary portfolios*, so the portfolios delivering $\min\{\mu\}$ and $\max\{\mu\}$.

8. Linear Regression Analysis

In this section we cover the most important assumptions underlying linear regression models and how to estimate such models in R. These lecture notes are not intended to replace classic econometric textbooks or lecture notes and are thus not as comprehensive on the technical details of regression models.

8.1. Univariate Linear Regression

We start with the univariate linear regression model of the form,

$$y_i = \alpha + \beta x_i + \epsilon_i, \qquad i = 1, \ldots, N, \tag{7}$$

where y_i dennotes the dependent variable, x_i the independent variable or *regressor* and ϵ_i the error term also called *resiudual*. The unknown parameters α and β are both scalars and we call α the intercept or constant (term) and β the regression coefficient. Note that α and β do not have a subscript i indicating that both variables are constant. The idea behind such models is that the value of y_i depends on the value of x_i and that this relationship is goverened by β. More precisely, we assume that the expected value of y_i equals $\alpha + \beta x_i$ and that deviations from the expected value are captured by the residual term ϵ_i. Thus, what we want to find are reasonable values for α and β. To find the variables of interest we first need to assume that the relationship stated in (7) indeed holds in the population, i.e. we did not formulate the wrong model to start with. We then estimate the values of the parameters by calibrating the model to available data for y_i and x_i. Note, however, that usually, we want to make a statement true for the whole population, while in general we do not observe data for the whole population, but can only draw on a sample of observations from the population – to be able to generalize from our estimated parameter values to the corresponding values in the whole population, we therefore need to ensure (or at least hope) that our sample is indeed representative of the population. When we estimate such a model on a sample we want to make clear that the estimated variables are *sample estimates* (rather than the true values valid for the whole population) and thus change notation

$$y_i = a + b x_i + e_i, \qquad i = 1, \ldots, N. \tag{8}$$

Here, a and b denote our sample estimates of the true population parameters α and β. Of course for an ever growing sample size we have that $a \to \alpha$, $b \to \beta$ and $e_i \to \epsilon_i$. Since we assume that the model is true in the population we want to choose a and b in such a way, that $a + bx_i$ best fits the data. In other words, we want that the error term is as small as possible. To achieve that we make use of the least squares criterion, i.e. we minimize the sum of squared errors.[8]

$$\min \sum_{i=1}^{N} e_i^2 = \sum_{i=1}^{N} (y_i - a - b x_i)^2 \tag{9}$$

[8] Think about why we want minimizing the sum of squared errors and not simply the sum of errors. Draw a line and dots that lie above and under it, measure their distance to the line (these are the errors) and compute the sum of of squared error and the sum of errors...what do you notice?

To minimize this expression we differentiate w.r.t. to the variables of interest (a and b) and set the derivative equal zero. In this way we find the values of the variables of interest that minimize (9).

$$\frac{\partial \sum_{i=1}^{N} (y_i - a - bx_i)^2}{\partial a} = 0 \implies a = \bar{y} + b\bar{x}$$
$$\frac{\partial \sum_{i=1}^{N} (y_i - a - bx_i)^2}{\partial b} = 0 \implies b = \frac{\text{Cov}(x, y)}{\text{Var}(x)},$$

where $\bar{y}$ is the sample mean of y and $\bar{x}$ the sample mean of x. A detailed derivation of this result can be found in the appendix. We will cover the assumptions underlying such models, e.g. that the relationship between y and x is indeed linear, in the next subsection as they are largely equivalent to the assumptions in a multivariate case. In `R` you can of course use the functions `mean`, `cov` and `var` to estimate a and b. But if you want to know something about their statistical significance you also need to compute the standard errors and corresponding test statistics. Luckily, `R` provides the function `lm` that will fit uni- and multivariate linear models and simultaneously computes standard errors and test statistics for the model parameters as well as the model R^2 and F-statistic.

8.2. Multivariate Linear Regression

In the model in equation 7, the dependent variable y is related to a single explanatory variable x, permitting us to model how a change in x influences y (in expectation). Now, we turn to the case where the dependent variable y is influenced by multiple explanatory variables at the same time. The multivariate linear regression model has the following form

$$y = X\beta + \epsilon, \tag{10}$$

where y again denotes the dependent variable. $y = (y_1, \ldots, y_N)'$ is a $(N \times 1)$ vector containing N observations of the dependent variable. Since y is the dependent variable, i.e. the variable that we want to explain with the model, y is also called and endogenous variable. X is the $(N \times K + 1)$ matrix containing the intercept and K regressors, also called the design matrix, β is the $(K + 1 \times 1)$ vector of coefficients corresponding to the intercept and the K regressors, and ϵ is the vector of errors/residuals. The $(N \times K + 1)$ design matrix X has the following form

$$X = \begin{pmatrix} 1 & x_{11} & x_{12} & \ldots & x_{1K} \\ 1 & x_{21} & x_{22} & \ldots & x_{2K} \\ \vdots & & & & \\ 1 & x_{N1} & x_{N2} & \ldots & x_{NK} \end{pmatrix}.$$

The columns of the design matrix correspond to different independent variables. Notice that we moved the intercept into the design matrix (the first column).[9] There are six main assumptions underlying linear regression models

[9] We could also write down the model as $y_i = \alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \epsilon_i$, but the matrix notation is more convenient.

1. The relationship between dependent and independent variables is linear and additive.

2. The independent variables are also independent of each other. This means that all variables vectors $x_k, k = 1, \ldots, K$ are linearly independent of each other and thus X has full rank.

3. The independent variables are exogenous to the model. So, we cannot use y to predict any x_k (in a non trivial way). This assumption holds if $\mathbb{E}[\epsilon|X] = 0$.

4. Homoscedasticity of the conditional variance of the error term. This means that the residual (error) variance is constant across observations $\rightarrow \text{Var}(e_i|X) = \sigma^2, i = 1, \ldots, N$.

5. The conditional covariance of the error term is zero. $\text{Cov}(e_i, e_j|X) = 0, i \neq j \implies \text{Var}(e|X) = \sigma^2 I$, where I denotes the $(N \times N)$ identity matrix.

6. $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$, the zero mean follows from $\mathbb{E}[\epsilon|X] = 0 \rightarrow \mathbb{E}[\mathbb{E}[\epsilon|X]] = \mathbb{E}[\epsilon] = 0$.

As in the univariate case we also find the coefficients of interest by applying the least squares ciriterion,

$$\min e'e = \min(y - Xb)'(y - Xb) = y'y - b'X'y - y'Xb + b'X'Xb = y'y - 2b'X'y + b'X'Xb.$$

Note that we again changed to "sample estimate notation". This minimization problem becomes straight forward to solve because of the matrix notation:

$$\frac{\partial e'e}{\partial b} = -2X'y + 2X'Xb = 0 \implies b = (X'X)^{-1} X'y$$

Here we used the assumption that X has full rank and thus $X'X$ is invertible! We need all the other assumption for the computation of standard errors and test statistics. We won't go into detail here but rather want to empirically demonstrate what happens if some assumptions are violated.

First let us start with a univariate model that does not violate any assumption.

```
true_b<-1.5
true_a<-1
x<-rnorm(n = 1000, mean = 3, sd = 4)
errors<-rnorm(n = 1000, mean = 0 , sd = 2)
y<-true_a + true_b*x + errors

model1<-lm(y~x)
summary(model1)

##
## Call:
## lm(formula = y ~ x)
##
```

```
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.8982 -1.3136  0.0304  1.2529  5.8073
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.14727    0.07799   14.71   <2e-16 ***
## x            1.47307    0.01587   92.83   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.963 on 998 degrees of freedom
## Multiple R-squared:  0.8962,Adjusted R-squared:  0.8961
## F-statistic:  8617 on 1 and 998 DF,  p-value: < 2.2e-16
```

We see both coefficients are highly significant and are close to there true values. Let's look at a model with two explanatories

```
true_b<-c(1,1.5,-0.4)
x0<-rep(1, 1000)
x1<-rnorm(n = 1000, mean = 3, sd = 4)
x2<-runif(n = 1000, min = -5, max = 5)
X<-cbind(x0,x1,x2)
errors<-rnorm(n = 1000, mean = 0 , sd = 2)
y<-X%*%true_b + errors

model2<-lm(y~X-1)
## we need to write -1 to tell lm that we have already
## incorporated an intercept in our design matrix X
summary(model2)

##
## Call:
## lm(formula = y ~ X - 1)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.4587 -1.3171  0.0777  1.4123  6.7176
##
## Coefficients:
##     Estimate Std. Error t value Pr(>|t|)
## Xx0  1.10764    0.07909   14.01   <2e-16 ***
## Xx1  1.49086    0.01632   91.34   <2e-16 ***
## Xx2 -0.42174    0.02285  -18.46   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.054 on 997 degrees of freedom
```

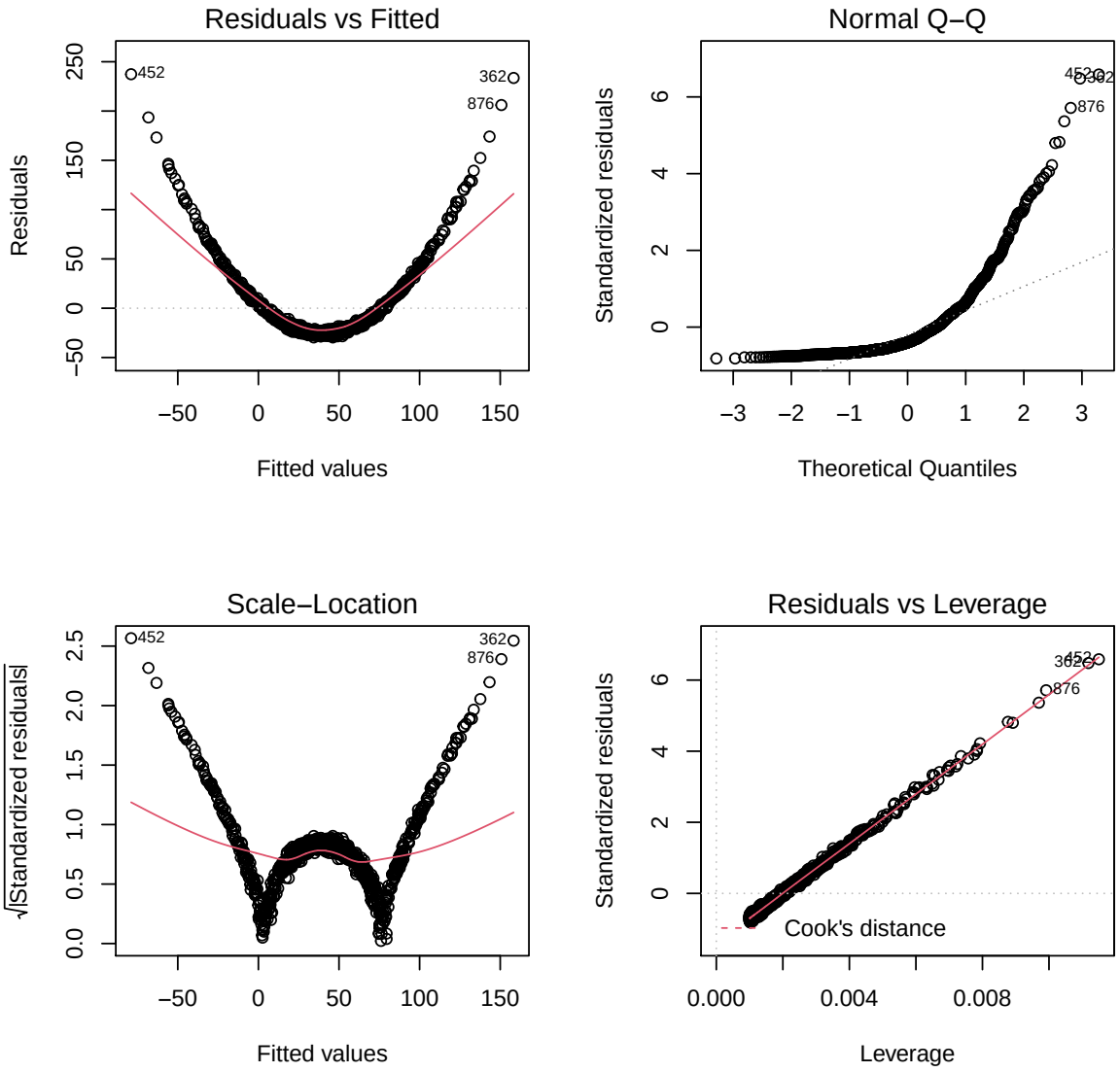
```
## Multiple R-squared:  0.9379,Adjusted R-squared:  0.9377
## F-statistic:  5018 on 3 and 997 DF,  p-value: < 2.2e-16
```

Also in this case all coefficients are highly significant and close to their true values. Let mess with a few assumption. We start with making the relationship non linear

```
true_a<-1
true_b<-1.5
x1<-rnorm(n = 1000, mean = 3, sd = 4)
errors<-rnorm(n = 1000, mean = 0 , sd = 2)
y<-true_a + true_b*x1^2 + errors

model3<-lm(y~x1)
summary(model3)

##
## Call:
## lm(formula = y ~ x1)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -29.65 -22.61 -14.52   8.19 237.33
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  13.1226     1.4290   9.183   <2e-16 ***
## x1            8.9536     0.2782  32.184   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 36.25 on 998 degrees of freedom
## Multiple R-squared:  0.5093,Adjusted R-squared:  0.5088
## F-statistic:  1036 on 1 and 998 DF,  p-value: < 2.2e-16
```

We see that the estimates for the intercept and the coefficient of x_1, β_1 are not even close to their true values. Since, in reality we do not know the real value of the coefficients how can we tell that something is wrong? R provides us with a tool usefull to check the linear regression modelling assumptions:

```
par(mfrow=c(2,2))
plot(model3)
par(mfrow=c(1,1))
```

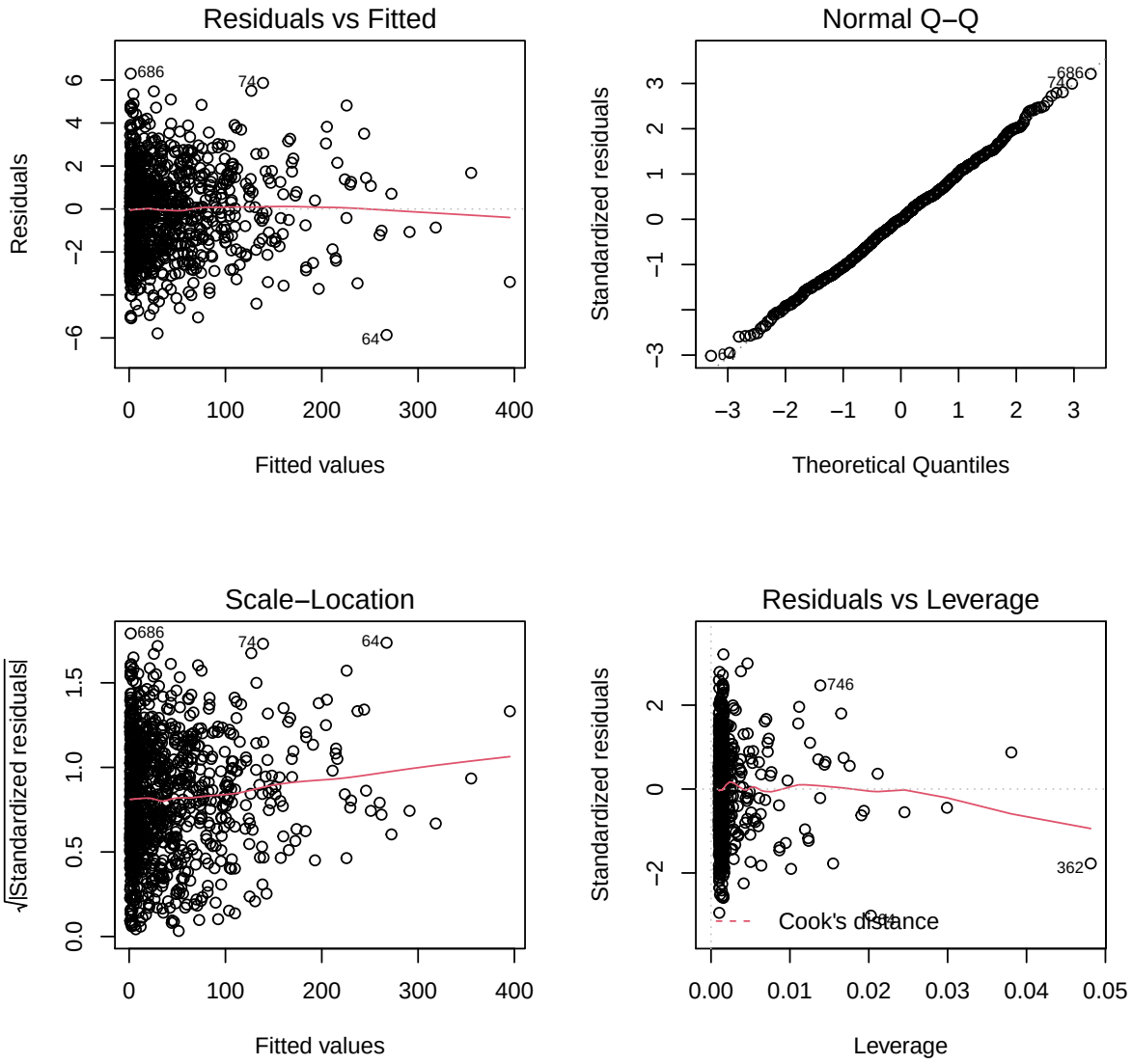


Take a look at the first plot — residuals vs fitted — it shows the fitted values, i.e. the values of y the estimated model predicts, plotted against the model errors. Ideally, these would be zero or randomly scattered around. However, you can cleary see a shape in this plot. This indicates that your modeled relationship in reality is not linear. Also, outliers are indicated by their number in this plot. The second plot (top right) indicates that the errors are not normally distributed: On the a-axis, theoretical quantiles of a true normal distribution are drawn – if the residuals from the estimated regression model are indeed normal (one of the assumptions stated above), the quantiles of the residuals, drawn on the y-axis, should correspond $1:1$ to those of the true normal distibution, leading to all points lying on a straigt line. In our example, however, we see that quantiles in both tails of the distribution of residuals do not correspond to the quantiles of an normal distribution, leading the points to be drawn far off the straigt dotted line. Usually, when the errors are not normally distributed, you can improve the model by transforming the variables in a non-linear manner given that the other assumptions hold. Let's see what happens if we specify the model correctly:

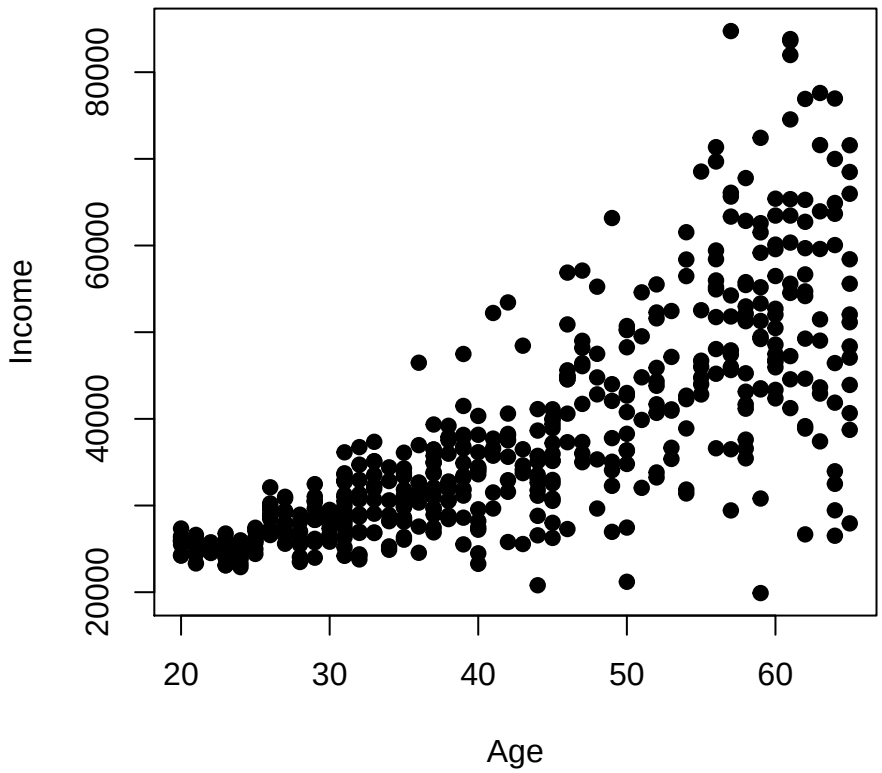
```r
true_a<-1
true_b<-1.5
x1<-rnorm(n = 1000, mean = 3, sd = 4)
errors<-rnorm(n = 1000, mean = 0 , sd = 2)
y<-true_a + true_b*x1^2 + errors
x1<-x1^2
model3<-lm(y~x1)
summary(model3)
```

```
##
## Call:
## lm(formula = y ~ x1)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.8605 -1.4002 -0.0028  1.2642  6.2972
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.032769   0.078172   13.21   <2e-16 ***
## x1          1.499072   0.001801  832.27   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.963 on 998 degrees of freedom
## Multiple R-squared:  0.9986,Adjusted R-squared:  0.9986
## F-statistic: 6.927e+05 on 1 and 998 DF,  p-value: < 2.2e-16
```

```r
par(mfrow=c(2,2))
plot(model3)
par(mfrow=c(1,1))
```



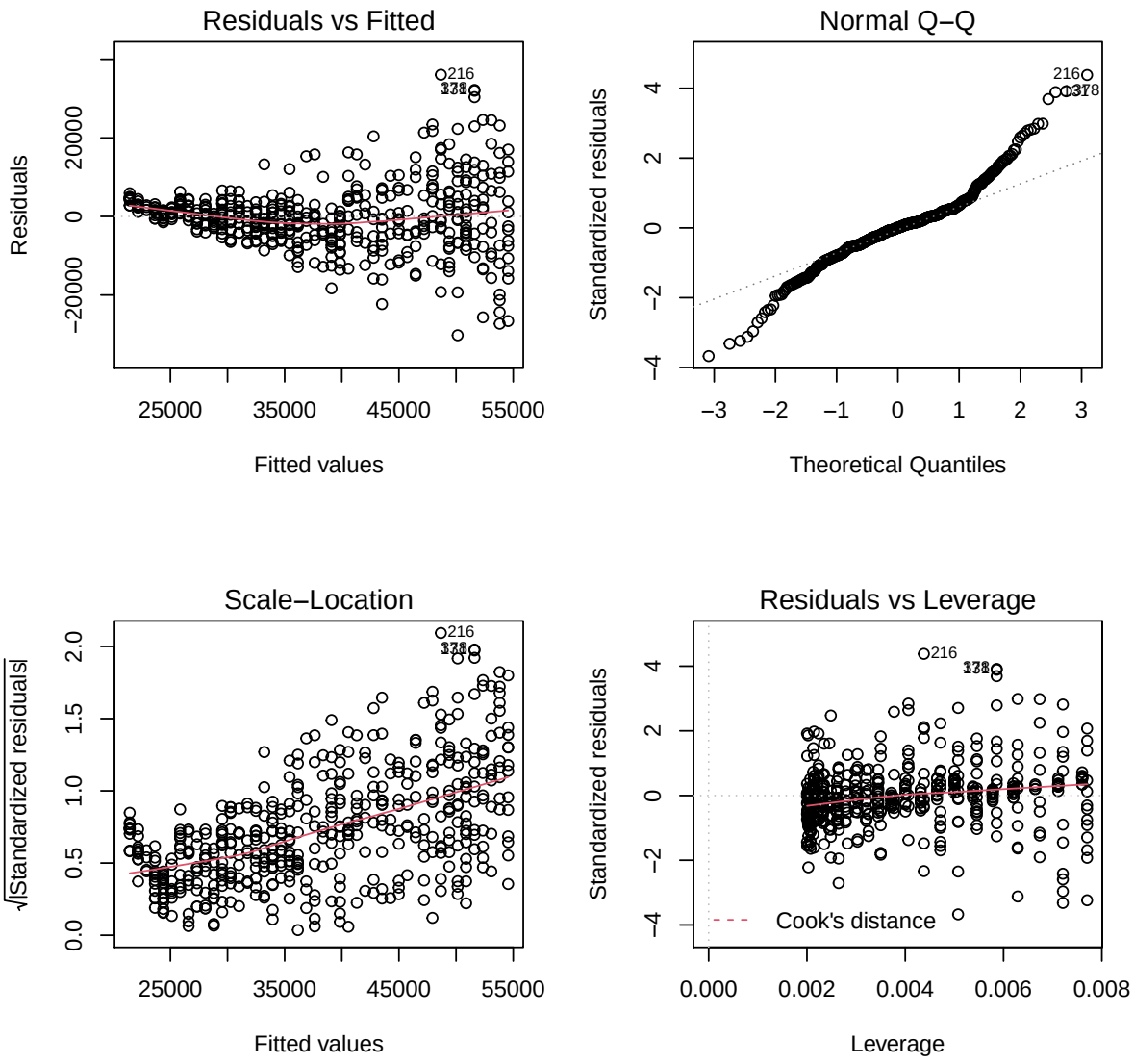
Both residual vs. fitted and normal q-q plot know look unsuspicious! Let's see what happens if the assumption of homoskedasticity is violated. Consider the following data on yearly income and age.




```
model4<-lm(income~age)
summary(model4)

##
## Call:
## lm(formula = income ~ age)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -30243  -4143     31   3197  36076
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  6720.66    1225.77   5.483 6.67e-08 ***
## age           735.94      27.57  26.695  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 8252 on 498 degrees of freedom
## Multiple R-squared:  0.5886,Adjusted R-squared:  0.5878
## F-statistic: 712.6 on 1 and 498 DF,  p-value: < 2.2e-16
```

An important thing to note is that you won't see any evidence of the violation of this assumption in the summary output. We again have to take a look at the plots!

```
par(mfrow=c(2,2))
plot(model4)
par(mfrow=c(1,1))
```



We see that the residual vs. fitted plot is funnel-shaped, a clear indication that heteroscedasticity is present in the error term. Also the sace-location plot should show no apparent pattern in the presence of homoscedasticity. The values of the coefficients themselves are unaffected by this phenomenon. However, the problem with heteroscedasticity in the residuals is that one under- or overestimate standard errors and thus, one could wrongly asses statistical significance. In such cases, it is therefore always advisible to *adjust* the standard errors for heteroscedasticity. One can do this by using White's heteroscedasticity consistent standard errors. In `R`, we can use the library `sandwhich`

to compute the robust covariance matrix estimator and then use the function `coeftest` from the library `lmtest` to compute the adjusted test statistic.

```
# Load libraries
library("sandwich")
library("lmtest")
covestimator<-vcovHC(model4, type = "HC0")
coeftest(x = model4, vcov. = covestimator)

##
## t test of coefficients:
##
##             Estimate Std. Error t value  Pr(>|t|)
## (Intercept) 6720.659   1096.619  6.1285 1.805e-09 ***
## age          735.941     32.285 22.7949 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As you can see, the coefficients are the same, but the standard errors and t-statistics have changed. Nevertheless, age is still significant when predicting income! We can also use a Breusch Pagan test to check for heteroskedasticity. The library `lmtest` provides us with the function `bptest`, that tells us if the error variance is homoskedastic (not significant test result) or heteroskedastic (significant test result).

```
bptest(model4)

##
##  studentized Breusch-Pagan test
##
## data:  model4
## BP = 91.007, df = 1, p-value < 2.2e-16
```

The last of the four plots — residuals vs. leverage — shows you the influence each observation has on the OLS estimate. Highly influential points are marked in the plot — for more information look up *Cook's distance* online. There is really no rule of thumb here, the best thing is to check the validality of highly influential points by hand.

Lastly, a more financial markets related example. We can use multiple univariate regression to compute industry betas by regressing excess industry returns on excess market returns. For this we downloaded the five industry protfolios from Kenneth R. French's website[10], the market excess return as well as the risk free rate.

```
m_cnsmr<-lm(industries[,1]~market)
m_manuf<-lm(industries[,2]~market)
m_hitec<-lm(industries[,3]~market)
m_hlth<-lm(industries[,4]~market)
m_other<-lm(industries[,5]~market)
betas<-c(m_cnsmr$coefficients[2],m_manuf$coefficients[2],
```

[10] https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html

```
         m_hitec$coefficients[2],m_hlth$coefficients[2],
         m_other$coefficients[2])
names(betas)<-c("Cnsmr", "Manuf", "Hitec", "Hlth", "Other")
betas

##     Cnsmr     Manuf     Hitec      Hlth     Other
## 0.9254662 0.9892297 0.9496081 0.8361439 1.1223226
```

We also want to asses whether or not these betas are significant and report 95% condifdence intervals. But before we can do that we need to check wheter we can use the standard errors provided to us by `lm` or if we need to correct them for heteroskedasticity.

```
bptest(m_cnsmr)

##
##  studentized Breusch-Pagan test
##
## data:  m_cnsmr
## BP = 16.552, df = 1, p-value = 4.734e-05

bptest(m_manuf)

##
##  studentized Breusch-Pagan test
##
## data:  m_manuf
## BP = 0.043432, df = 1, p-value = 0.8349

bptest(m_hitec)

##
##  studentized Breusch-Pagan test
##
## data:  m_hitec
## BP = 5.9259, df = 1, p-value = 0.01492

bptest(m_hlth)

##
##  studentized Breusch-Pagan test
##
## data:  m_hlth
## BP = 0.0067739, df = 1, p-value = 0.9344

bptest(m_other)

##
##  studentized Breusch-Pagan test
##
## data:  m_other
## BP = 61.428, df = 1, p-value = 4.593e-15
```

According to the Breusch-Pagan test all models, except for the manufacturing and health care industry models, suffer from heteroskedasticity in the residuals and thus we need to adjust the respective standard errors.

```
adj_se1<-coeftest(x = m_cnsmr,vcov. = vcovHC(m_cnsmr, type = "HC0"))
adj_se2<-coeftest(x = m_hitec,vcov. = vcovHC(m_hitec, type = "HC0"))
adj_se3<-coeftest(x = m_other,vcov. = vcovHC(m_other, type = "HC0"))
t.stats<-c(adj_se1[6], summary(m_manuf)$coefficients[6],
           adj_se2[6], summary(m_hlth)$coefficients[6],
           adj_se3[6])
names(t.stats)<-c("Cnsmr", "Manuf", "Hitec", "Hlth", "Other")
ci<-rbind(paste0("Beta Cnsmr 95% CI: [",
                 round(m_cnsmr$coefficients[2]-1.96*adj_se1[4],2),",",
                 round(m_cnsmr$coefficients[2]+1.96*adj_se1[4],2),"]"),
          paste0("Beta Manuf 95% CI: [",
                 round(m_manuf$coefficients[2]-1.96*
                           summary(m_manuf)$coefficients[4],2),","
                 ,round(m_manuf$coefficients[2]+1.96*
                            summary(m_manuf)$coefficients[4],2),"]"),
          paste0("Beta HiTec 95% CI: [",
                 round(m_hitec$coefficients[2]-1.96*adj_se2[4],2),","
                 ,round(m_hitec$coefficients[2]+1.96*adj_se2[4],2),"]"),
          paste0("Beta Hlth 95% CI: [",
                 round(m_hlth$coefficients[2]-1.96*
                           summary(m_hlth)$coefficients[4],2),","
                 ,round(m_hlth$coefficients[2]+1.96*
                            summary(m_hlth)$coefficients[4],2),"]"),
          paste0("Beta Other 95% CI: [",
                 round(m_other$coefficients[2]-1.96*adj_se3[4],2),","
                 ,round(m_other$coefficients[2]+1.96*adj_se3[4],2),"]"))
t.stats

##     Cnsmr     Manuf     Hitec      Hlth     Other
##  55.00459 110.36807  43.22193  45.69141  31.78327

ci

##      [,1]
## [1,] "Beta Cnsmr 95% CI: [0.89,0.96]"
## [2,] "Beta Manuf 95% CI: [0.97,1.01]"
## [3,] "Beta HiTec 95% CI: [0.91,0.99]"
## [4,] "Beta Hlth 95% CI: [0.8,0.87]"
## [5,] "Beta Other 95% CI: [1.05,1.19]"
```

We see that all estimated betas are highly significant and given the non overlapping confidence intervals we conclude that the e.g. manufacturing industry is exposed to more systematic risk than the health care industry.

8.3. Additional Example: Fama & French Three Factor Model

In the following, we use multivariate linear regressions to estimate the parameters of a Fama & French three factor model. For this purpose, we consider 12 industry portfolios and estimate the exposure of their excess returns (*excess return*, in the following, refers to the return in excess of the risk-free rate) to the risk factors included in the Fama & French three factor model for each industry i:

$$(r_{i,t} - r_{f,t}) = \alpha + \beta_{\text{Mkt},i}(r_{\text{Mkt},t} - r_{f,t}) + \beta_{\text{SMB},i}\text{SMB}_t + \beta_{\text{HML},i}\text{HML}_t + \epsilon_{i,t}. \tag{11}$$

Essentialy, equation (11) says that the excess industry return, $(r_{i,t} - r_{f,t})$, is a function of the excess market return, $(r_{\text{Mkt,t}} - r_{f,t})$, the small minus big risk factor, SMB, and the high minus low risk factor, HML. Now, if $\alpha = 0$, then the industry excess return is indeed a function of just these three risk factors (market, SMB and HML). The idea here is similar to the idea of the CAPM: The excess return of a stock is tied to its exposure to different sources of risk that carry a risk premium, and corresponds to the sum of the associated risk premia – only that in this case, we do not only consider exposure to the market risk (as in CAPM), but there are also two additional risk premia. But let's take it slow and start with some more intution for the CAPM, then also the idea behind the Fama & French factors will become more clear:

The CAPM states that a stock return is a function of the market return and the risk-free rate and thus the excess stock return is a function of the excess market return and nothing else. The argument is that stocks carry two types of risk: *systematic* and *idiosyncratic* risk, also referred to as undiversifiable and diversifiable risk respectively. You can think of it in this way: Systematic risk is undiversifiable because it affects the *whole* economy, e.g. earthquakes, employment shocks, adverse government policies,.... Idiosyncratic risk, by contrast, only affects a *single* firm, e.g. production machinery is destroyed in an accident, bad management decisions,.... Because these latter events only affect a single firm, idiosyncratic risk is diversifiable in the sense that if you hold a portfolio of many stocks, adverse events affecting only single firms will affect your overall portfolio return only very, very slightly: on the level of your portolio's return, particularly positive or negative events occurring only to single firms more or less "average out". For example, consider holding a portfolio of 100 firms where each firm delivers an *expected* return of 5% every month, so your monthly expected portfolio return is 5%. If some firms deviate to the positive while others deviate to the negative from their expeted return due to idiosyncratic events relevant only to single firms, your portfolio return will still be reasonably close to 5%, even if individual firms deviated more drastically, making, for example, -5%, or +15%. Crucially, it is only idiosyncratic risk that can be "diversified away". Systematic risk, as we said, affects all firms (the whole economy) at the same time, so holding a diversified portfolio of many stocks still does not lead to systematic risk "averaging out". It is for this reason that exposure to systematic risk is associated with compensation in the form of a risk premium. Risk averse investors dislike risk, so if investments bear systematic risk, asset prices are accordingly lower, leading to a higher expected return. Why does only systematic risk offer a positive risk premium, while there is no compensation for idiosyncratic risk? Because idiosyncratic risk can be avoided at no cost, simply by holding a whole portfolio instead of just a single stock – so risk averse investors do not require additional compensation, since they can can get rid of this form of risk simply through portfolio diversification. Systematic risk affects every firm, and is

therefore the risk inherent in the *market portfolio*, i.e. the portfolio proportional to the whole universe of stocks available to every investor – thus a) the return on the market portfolio is driven by systematic risk, carrying the risk premium of systematic risk, and b) systematic risk is also called *market risk*. In the CAPM model world, stocks/firms have different exposures to that market risk, indicated by their β_{Mkt}, the market beta, which measures to what extent the return of a single stock co-moves with the market. After the CAPM was introduced (and even before that) more and more research popped up that suggested that small firms deliver higher returns than large firms and that firms with high book-to-market (BTM) ratios outperformed firms with low BTM ratios even when controlling for market risk, i.e. stock's exposure to systemic risk. Fama & French added two additional sources of risk (SMB and HML) to the CAPM model measuring these two phenomena. Each of these two additional types of risk carry their own risk premia. Simply put, SMB measures the outperformance of small stocks compared to large stocks and HML the outpoerformance of high BTM compared to low BTM stocks. Therefore, the Fama & French three factor model states that a stock's excess return is a function of the market excess return and the differences in return between small and large as well as high and low BTM stocks.

In the following, we estimate the Fama & French three factor model separately for each industry i, based on returns for 12 industry portfolios obtained from the website of Kenneth R. French:

$$(r_{i,t} - r_{f,t}) = \alpha + \beta_{\text{Mkt},i}(r_{\text{Mkt},t} - r_{f,t}) + \beta_{\text{SMB},i}\text{SMB}_t + \beta_{\text{HML},i}\text{HML}_t + \epsilon_{i,t}.$$

```
## We downloaded the 12 industry portfolios and the factor data
# from Kenneth French's website
head(industries.excess)
```

```
##            NoDur Durbl Manuf Enrgy Chems BusEq Telcm Utils Shops  Hlth Money
## 1926-07-01  1.23 15.33  3.45 -1.40  7.79  2.94  0.61  6.82 -0.11  1.55  0.15
## 1926-08-01  3.72  3.43  2.17  3.22  4.89  1.72  1.92 -1.94 -0.96  4.00  4.21
## 1926-09-01  0.91  4.57 -0.30 -3.62  5.07 -0.57  2.18  1.81 -0.02  0.46 -1.46
## 1926-10-01 -1.56 -8.55 -3.48 -1.10 -4.87 -5.70 -0.43 -2.95 -2.61 -0.89 -5.48
## 1926-11-01  4.89 -0.50  3.51 -0.30  4.80  4.48  1.32  3.40  6.12  5.11  1.93
## 1926-12-01  0.54  9.61  3.44  2.54  4.85 -2.06  1.71 -0.45  0.34 -0.17  2.40
##            Other
## 1926-07-01  2.02
## 1926-08-01  4.12
## 1926-09-01  0.12
## 1926-10-01 -3.08
## 1926-11-01  1.79
## 1926-12-01  3.15
```

```
head(ff3_data)
```

```
##            market.excess   smb   hml   rf
## 1926-07-01          2.96 -2.30 -2.87 0.22
## 1926-08-01          2.64 -1.40  4.19 0.25
## 1926-09-01          0.36 -1.32  0.01 0.23
```

```
## 1926-10-01         -3.24  0.04  0.51 0.32
## 1926-11-01          2.53 -0.20 -0.35 0.31
## 1926-12-01          2.62 -0.04 -0.02 0.28

##First step
## Run 12 regression to get the factor exposures, i.e. betas for each industry
exposures<-list()
for(i in 1:12){ # i loops through industries
  exposures[[i]] <- lm(industries.excess[,i] ~ market.excess + smb + hml,
                       data=ff3_data)
}

#extract betas for every industry (sapply) and put them in a dataframe
exposures <- sapply(X = exposures, FUN = function(x)x$coefficients)
exposures <- as.data.frame(t(exposures))
colnames(exposures)[1] <- "alpha"
row.names(exposures) <- colnames(industries.excess)
exposures

##              alpha market.excess         smb         hml
## NoDur  0.18356361     0.7601379 -0.04696314  0.01052530
## Durbl -0.02319924     1.2210858  0.05165491  0.17496486
## Manuf -0.09395690     1.1432398  0.09245461  0.20277397
## Enrgy  0.02832584     0.8919307 -0.17245213  0.25122931
## Chems  0.12222847     1.0023703 -0.18786056 -0.01878392
## BusEq  0.08714360     1.2942042  0.08930626 -0.28407074
## Telcm  0.15485019     0.6975489 -0.13551861 -0.04658571
## Utils  0.03091051     0.7600551 -0.16391058  0.26485281
## Shops  0.13896425     0.9711596  0.07544532 -0.12060041
## Hlth   0.29825832     0.8826257 -0.08829864 -0.19176239
## Money -0.11686584     1.1276329 -0.06012604  0.30599489
## Other -0.26689917     1.0417103  0.22362651  0.28964344
```

A higher coefficient (for exposure to market risk, SMB or HML) means that an industry's returns are driven more strongly by exposure to the respective risk factor. For example, we can see that durables moves more strongly with the market in comparisison to telecommunications. Note that negative coefficients mean that an industry's excess returns move *against* the risk factor in questions, i.e. the industry performs particularly good when this sort of risk materializes. The α is the part of an industry's excess return not attributable to exposure to any of the three sources of risk considered in the model – it therefore represents the amount of average excess return per unit of time earned without taking any of these three risks. Finally, note that in this example, we only concentrated on coefficient estimates and the intuition behind them, without paying attention to statistical significance – in order to assess statistical significance of the estimates, once should check that the regression model assumptions are met and, if necessary, adjust standard errors appropriately, as shown further above in this chapter.

9. PCA – Application to the German sovereign yield curve

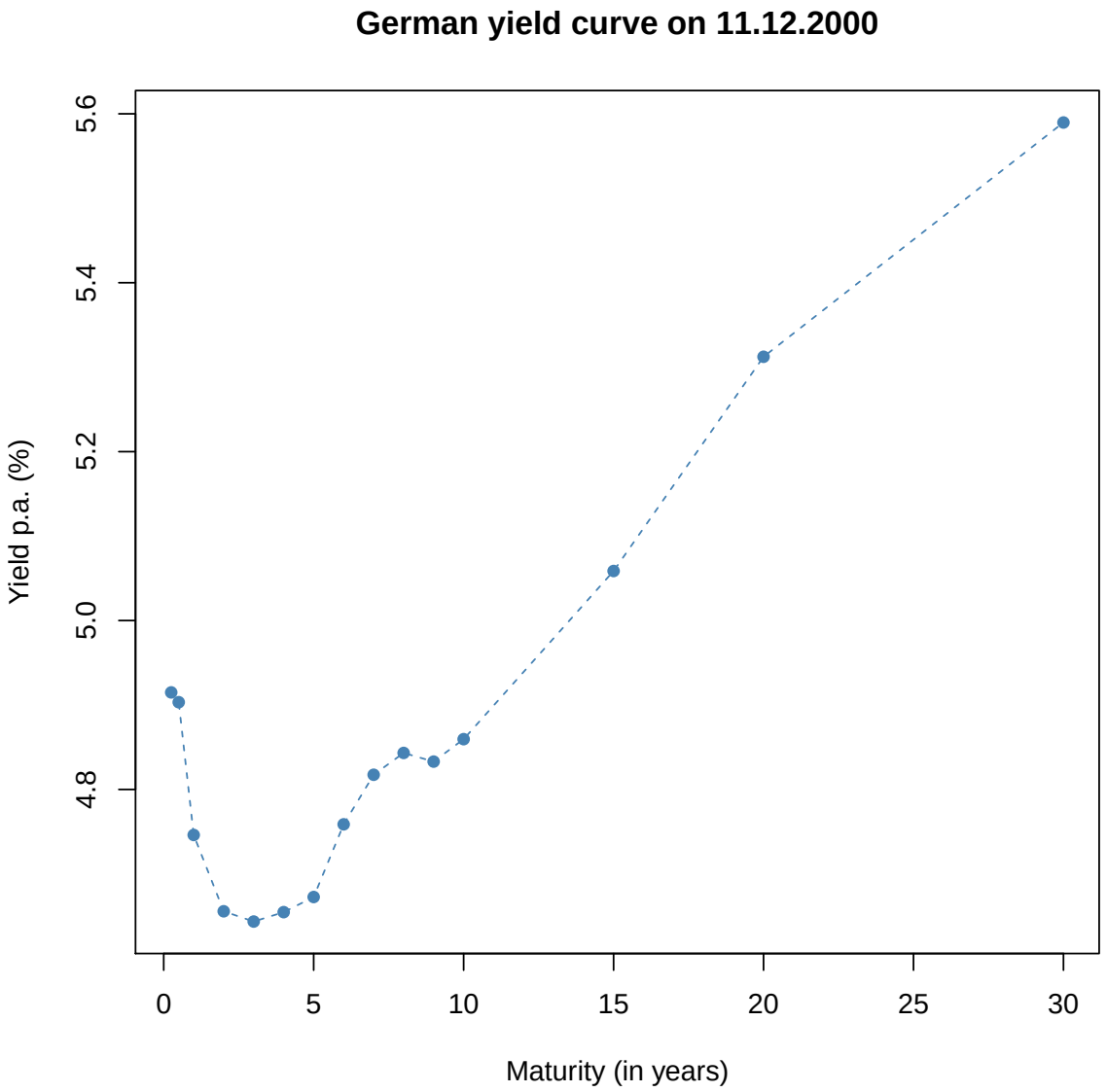
9.1. Overview of the data and the problem

We now revisit the data set on German sovereign bond yields already introduced in Section 1. Let us assign the data to a new object called `yields` and have a quick look:

```
yields <- german.interest
head(yields)

##         Date     X3M     X6M     X1Y     X2Y     X3Y     X4Y     X5Y     X6Y
## 1 11.12.2000 4.91491 4.90332 4.74611 4.65575 4.64359 4.65474 4.67264 4.75868
## 2 12.12.2000 4.91491 4.90622 4.74963 4.65641 4.64530 4.65257 4.67391 4.76458
## 3 13.12.2000 4.91395 4.90332 4.70384 4.59706 4.58216 4.58877 4.60987 4.70100
## 4 14.12.2000 4.90139 4.87723 4.68483 4.52651 4.55040 4.55694 4.57871 4.67291
## 5 15.12.2000 4.89752 4.87337 4.66248 4.50157 4.51175 4.50756 4.52840 4.62212
## 6 18.12.2000 4.88206 4.85597 4.61884 4.47767 4.47037 4.46584 4.48716 4.58597
##       X7Y     X8Y     X9Y    X10Y    X15Y    X20Y    X30Y
## 1 4.81739 4.84323 4.83299 4.85954 5.05866 5.31231 5.58972
## 2 4.82910 4.85977 4.85315 4.88013 5.07307 5.32375 5.59810
## 3 4.76851 4.80935 4.80022 4.83391 5.02300 5.27704 5.55546
## 4 4.73771 4.78923 4.78090 4.81444 4.99204 5.22764 5.48557
## 5 4.69189 4.74841 4.74071 4.77875 4.95540 5.19238 5.45185
## 6 4.65824 4.72371 4.72434 4.75643 4.94766 5.19411 5.46294
```

For every trading day, the data set contains the (annualized) yields on German bonds with different maturities, ranging from 3 months to 30 years. For example, if on December 11, 2000, one bought a German bond with a maturity of 2 years and held it until maturity, this investment promised a yield of 4.65575% per year. On the same date, one could have also acquired a 10 year bond with a yield to maturity of 4.85954%. The yields across all maturities for a given day are often called the *yield curve*, because they can conveniently be visualized by a curve. On a given day, plot the different maturities (in years) on the x-axis and the bonds' corresponding annual yields to maturity on the y-axis.

```
maturities <- c(3/12, 6/12, 1:10, 15, 20, 30)
plot(x=maturities, y=yields[1,-1],
     col="steelblue", type="o", pch=16, lty=2,
     xlab="Maturity (in years)", ylab="Yield p.a. (%)",
     main=paste0("German yield curve on ", yields[1,1]))
```

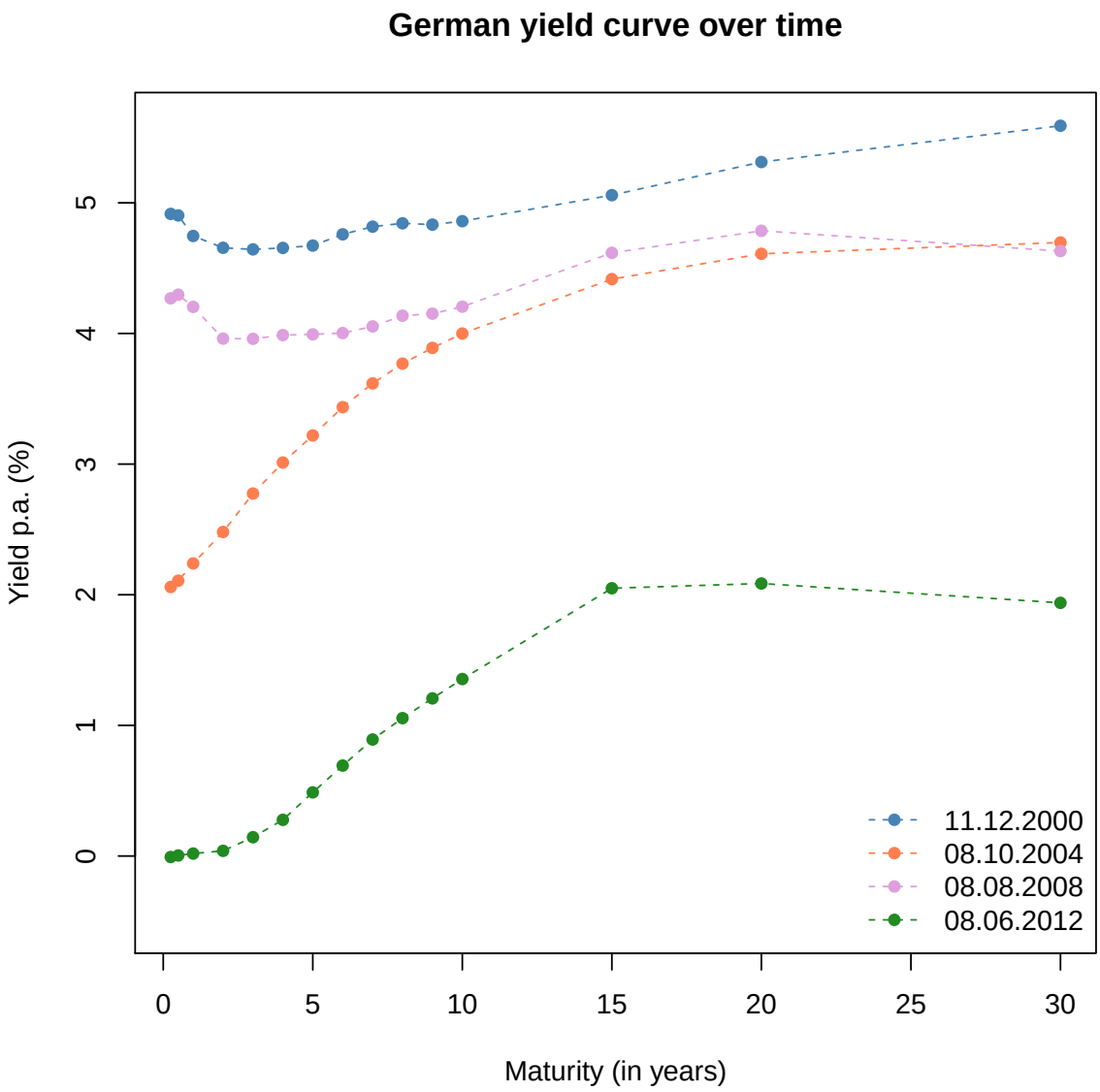



```
#type="o" plots points+line, pch controls point shape
#paste0 can be used to paste stuff together to a single character element
```

From the head of our data, we can already see that the yield on a German sovereign bond of any given maturity changes from day to day – in other words, the shape of the whole yield curve changes over time. These changes become even more apparent once we compare the yield curves of days further apart:

```
plot(x=maturities, y=yields[1,-1],
     ylim=c(-0.5, 5.6),
     col="steelblue", type="o", pch=16, lty=2,
     xlab="Maturity (in years)", ylab="Yield p.a. (%)",
     main="German yield curve over time")
lines(x=maturities, y=yields[1000,-1], type="o", pch=16, lty=2, col="coral")
lines(x=maturities, y=yields[2000,-1], type="o", pch=16, lty=2, col="plum")
lines(x=maturities, y=yields[3000,-1], type="o", pch=16, lty=2, col="forestgreen")
```

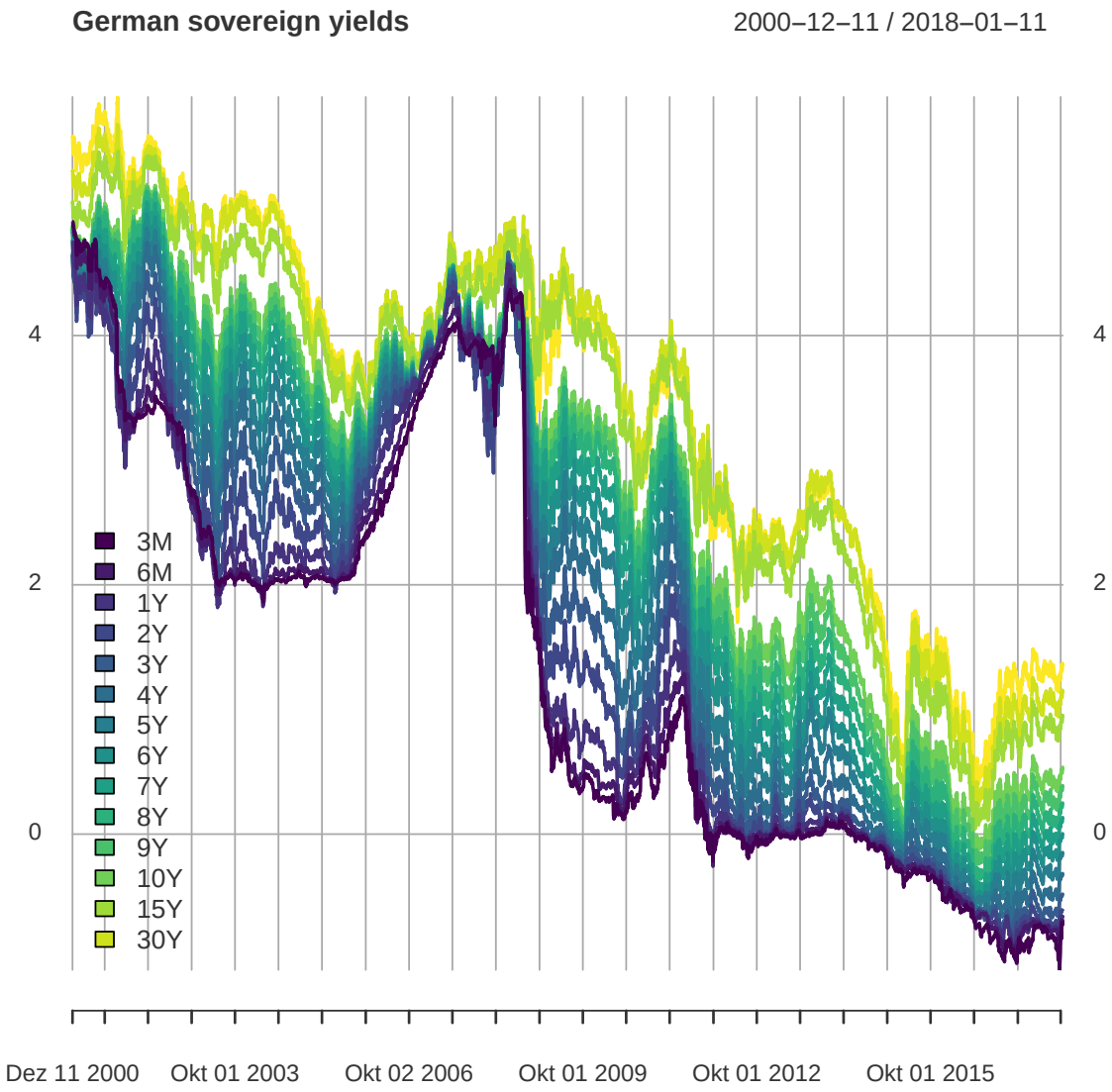
```
legend("bottomright", legend=yields$Date[c(1,1000,2000,3000)], bty="n",
       pch=16, lty=2, col=c("steelblue", "coral", "plum", "forestgreen"))
```



To examine this time-varying behaviour in more detail, we can also vizualize the yields on German sovereign bonds for every single trading day between December 2000 and January 2018:

```
#load viridis color package
library(viridis)
# convert to xts and plot
# note: viridis colors are accessed using the function viridis()
yields.xts <- xts(yields[,-1], order.by=as.Date(yields$Date, format="%d.%m.%Y"))
yield_plot <- plot(yields.xts, col=viridis(n=ncol(yields.xts)),
     main="German sovereign yields")
yield_plot <- addLegend("bottomleft",
          legend.names=c("3M", "6M", paste0(c(1:10,15,30),"Y")),
          fill=viridis(n=ncol(yields.xts)))
```

```
yield_plot
```



This last plot confirms our previous assessment that the yield curve indeed changes quite drastically over time – the absolute level of German sovereign yields as well as the relative level of short maturities (e.g. 3M) compared to long maturities (e.g. 30Y). Yet, at the same time, we also observe that there appears to exist some kind of co-movement between yields of different maturities – they do not evolve completely independent of each other.

The Problem: In many applications, it would be useful to have a simple summary of the evolution of the German sovereign yield curve, rather than the yields of 15 individual maturities, which sometimes can be a little overwhelming. For example, suppose that we want to include the "level of German sovereign yields" as an explanatory variable in a linear regression model[11]. Including 15 individual maturities and estimating all their

[11] For reasons beyond the scope of this course, in a real world application it might be necessary to include daily changes in yields rather than the daily level of yields in a linear regression model.

corresponding coefficients would unnecessarily inflate the complexity of the model – after all, the e.g. 3M and 6M maturities do not appear to evolve that differently from each other and would therefore provide more or less the same information. As another example, suppose you hold a portfolio of assets which pay risk-free cash flows over a horizon of the next 30 years. Using German sovereign yields as a measure for risk-free yields[12], you would like to know how the value of your portfolio reacts to changes to the risk-free yield curve. What happens if the yield curve shifts up by 100 basis points? What happens if the yield curve gets steeper (i.e. yields on long maturities increase relative to yields on short maturities)? Rather than considering changes at every single maturity of the yield curve, it might be simpler to come up with a summary measure for the overall level of yields (across all maturities) and the steepness of the yield curve, and relate the portfolio value to changes in these two summary quantities. How can we come up with a single or a low number of measures that meaningfully summarize the evolution of the German sovereign yield curve over time? One answer is *principal component analysis (PCA)*, which we introduce in the following.

9.2. PCA – intuition and implementation in `R`

Starting from a high-dimensional data set, the main purpose of PCA is to extract a small number of variables that still capture most of the variation found in the original data. Intuitively, for example the 3M and 6M maturity yields evolve almost identically over time, and one could therefore extract a single variable that simultaneously describes movements in both of them with reasonable accuracy (even though not perfectly). These "summary variables" that PCA extracts from the raw data are called *principal components (PCs)*. To summarize as much information as possible in a small number of variables, principal components satisfy the following criteria:

- Principal components are uncorrelated to each other – i.e. a principal component does not redundantly contain any information already captured in another principal component.
- The first principal component is chosen such that it captures as much variation across all variables in the raw data as possible. The second principal component is then chosen such that it captures as much as possible of the remaining variation in the original data not already captured by the first principal component. The third principal component in turn captures as much as possible of the variation in the original data not already captured by the first two principal components. This logic applies for any further (the fourth, sixth,...) principal components extracted from the raw data.

The question now is how to find principal components that satisfy these criteria. In the following, we describe the procedure by means of a practical application to our data of German sovereign yields. The exact reasons *why* this procedure turns out to be successful, i.e., the mathematical theory behind it, are beyond the scope of this course.[13] Instead, we would like to focus on developing sound intuition for what PCA can be used for, how to apply it correctly and how the outcome of PCA can be interpreted.

[12] This is commonly done in practice, since the probability that Germany will default on its sovereign debt is gerneally perceived as extremely low.

[13] We provide a take on an intuitive, geometric explanation in the appendix.

The goal. Our goal is to extract a small number of variables that reasonably well describe the evolution of yields across all 15 maturities contained in our data set on German sovereign yields. In other words, we would like to reduce the dimension of our data while retaining as much information as possible.

Step 1: Normalization of the raw data. Before conducting PCA, we have to ensure that our data fulfill the following two criteria, which are usually not satisfied ex ante but require transformation of the raw data. First, since PCA focuses exclusively on capturing the *variation* in the data (i.e. fluctuations in each variable around its mean, but not the mean itself), each variable in the raw data set first needs to be centered to zero (i.e. we subtract the mean of each variable – for the centered data, all variables then have a mean of zero). Second, each principal component tries to capture as much of the variation *across all variables* in the original (centered) data (not already captured by any previous principal component) as possible. In order for PCA to give equal weight to each variable in the raw data, all variables therefore have to be standardized to a common scale. This is particularly true if variables are measured in different units or on different scales. For example, consider a data set that measures the distances from one location, L, in Austria to various cities in different countries. The first variable measure the distance from L to all major cities in Austria in kilometers (km), the second variable measures the distance from L to all major cities in Germany in km and the third variable measure the distance form L to all major cities in France in *meters*. Now, the first two variables are comparable because their unit of measurement is the same. However, the third variable will have "larger numbers" as data since its unit of measurement is different. Without adjusting the unit of measurement we would always arrive at the conclusion that variable three has the largest variance regardless of where L actually is. As another example, consider that we have a data set with two variables measuring the distance between various places and Vienna. One variable captures the road distance to Vienna in meters. The other variable captures the time in hours it takes to get from one place to another by public transport. Without standardization, the meter distance variable would always have the larger variance. However, if we de-mean both variables and scale them such that their standard deviations both equal to one, we can actually compare the relative variation of road distance and time and therefore also the relative importance of these two variables. If all variables in the original data are measured on the same scale in the same units, scaling is optional (the data would still need to be centered in any case), but still common to ensure equal importance of all variables. In general, we therefore de-mean each variable in the raw data and divide by its standard deviation. Resulting standardized variables used for subsequent PCA analysis have a mean of 0 and a standard deviation of 1.

```
#1. drop first column containing dates
#2. standardize each column
yields_std <- apply(yields[,-1], 2, function(x) (x - mean(x))/sd(x))

# check mean of each standardized variable
# -> basically zero (just numerical imprecision)
colMeans(yields_std)

##           X3M           X6M           X1Y           X2Y           X3Y
```

```
## -2.654177e-17  3.963837e-17  1.620891e-17  3.137208e-18  6.934225e-17
##           X4Y           X5Y           X6Y           X7Y           X8Y
##  3.635177e-18 -2.594421e-17 -1.169979e-16  5.544891e-17 -3.241782e-17
##           X9Y          X10Y          X15Y          X20Y          X30Y
## -9.321989e-17  1.113460e-16 -6.423807e-17  1.150310e-17  1.663218e-17

# check standard deviation of each standardized variable
# -> exactly 1 for all variables
apply(yields_std, 2, sd)

##  X3M  X6M  X1Y  X2Y  X3Y  X4Y  X5Y  X6Y  X7Y  X8Y  X9Y X10Y X15Y X20Y X30Y
##    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1
```

Step 2: The `prcomp()` function. To perform PCA on the standardized data, `R` provides the function `prcomp()`. Note that the function itself also takes two additional arguments, `center` and `scale.`. Instead of manually centering and scaling the data in the previous step, one can also simply set these two arguments to `TRUE`.

```
# could omit last two arguments since we previously already scaled data by hand
pca <- prcomp(x=yields_std, center=TRUE, scale.=TRUE)
```

The function performs the following transformation on the data:

$$\mathbf{D} = \mathbf{X}\mathbf{\Lambda}' \tag{12}$$

D denotes the $(N \times K)$ matrix containing the standardized original data (provided to the function). In our case, N corresponds to the number of trading days, and K corresponds to the number of distinct maturities in our raw data. X is an $(N \times K)$ matrix containing the K principal components in its columns.[14] The leftmost column contains the first principal component, which captures the largest share of the raw data variance. The rightmost column is the last principal component and captures the least variance. Λ' is the $(K \times K)$ transpose of the so-called loadings matrix that relates maturities to the principal components. Λ is also called the rotation matrix.[15] In expanded notation:

$$\underbrace{\begin{pmatrix} d_1^{3M} & d_1^{6M} & d_1^{1Y} & \dots & d_1^{30Y} \\ d_2^{3M} & d_2^{6M} & d_2^{1Y} & \dots & d_2^{30Y} \\ \vdots & \vdots & \vdots & & \vdots \\ d_N^{3M} & d_N^{6M} & d_N^{1Y} & \dots & d_N^{30Y} \end{pmatrix}}_{\mathbf{D}(N\times K)} = \underbrace{\begin{pmatrix} pc_1^1 & pc_1^2 & pc_1^3 & \dots & pc_1^{15} \\ pc_2^1 & pc_2^2 & pc_2^3 & \dots & pc_2^{15} \\ \vdots & \vdots & \vdots & & \vdots \\ pc_N^1 & pc_N^2 & pc_N^3 & \dots & pc_N^{15} \end{pmatrix}}_{\mathbf{X}(N\times K)} \underbrace{\begin{pmatrix} \lambda'_{1,1} & \lambda'_{1,2} & \lambda'_{1,3} & \dots & \lambda'_{1,15} \\ \lambda'_{2,1} & \lambda'_{2,2} & \lambda'_{2,3} & \dots & \lambda'_{2,15} \\ \vdots & \vdots & \vdots & & \vdots \\ \lambda'_{15,1} & \lambda'_{15,2} & \lambda'_{15,3} & \dots & \lambda'_{15,15} \end{pmatrix}}_{\mathbf{\Lambda}'(K\times K)}$$

Note that $\mathbf{\Lambda}$ has the special property that $\mathbf{\Lambda}'\mathbf{\Lambda} = \mathbf{\Lambda}\mathbf{\Lambda}' = \mathbf{I}$, i.e. matrix multiplication of $\mathbf{\Lambda}$ with its transpose results in the identity matrix.[16] For this reason, we can multiply

[14] There exist as many principal components as variables. However, we are only interested in "the most important ones".

[15] The illustrative example in the appendix should shed some more light on the variable names used in PCA.

[16] We also say Λ is an orthogonal matrix.

from the right with $\mathbf{\Lambda}$ on both sides of equation 12 to obtain:

$$\mathbf{X} = \mathbf{D}\mathbf{\Lambda} \tag{13}$$

This notation corresponds to the output provided by `prcomp()`. The object returned by `prcomp()` contains the principal components (`x`), the loadings matrix (`rotation`), the standard deviation of each PC (`sdev`), the variable means of the original data (`center`) and an indicator if the data was scaled or not (`scale`). Let us have a look at the output.

```
head(pca$x)
```

```
##             PC1      PC2        PC3        PC4         PC5         PC6
## [1,] -6.081497 1.037413 -0.3203604 -0.1321757 -0.07321641 -0.02189511
## [2,] -6.100908 1.023965 -0.3227673 -0.1300188 -0.06484933 -0.02875926
## [3,] -5.985754 1.047959 -0.3541983 -0.1319064 -0.05000979 -0.02381900
## [4,] -5.906152 1.061385 -0.3457994 -0.1211566 -0.01819931 -0.01740306
## [5,] -5.823193 1.085520 -0.3659894 -0.1187270 -0.01475329 -0.02127383
## [6,] -5.767962 1.067511 -0.3950289 -0.1233175 -0.01955004 -0.02262263
##               PC7         PC8        PC9        PC10         PC11       PC12
## [1,] -0.05152190 0.015000417 0.03459781 -0.02169413 -0.007507712 0.01779884
## [2,] -0.04893381 0.013680752 0.03323219 -0.02370706 -0.005252544 0.01886780
## [3,] -0.04868153 0.013476358 0.02713000 -0.02859567 -0.007448135 0.01807423
## [4,] -0.04599069 0.007847727 0.01667908 -0.01498554  0.004125814 0.02034889
## [5,] -0.04242186 0.008119698 0.01651371 -0.02109158  0.002839458 0.02067564
## [6,] -0.03979770 0.009793494 0.01919831 -0.03188889 -0.001098131 0.01759916
##              PC13        PC14          PC15
## [1,] 0.014628108 0.002416297 -0.0003720605
## [2,] 0.013456331 0.003100471 -0.0001402796
## [3,] 0.013564071 0.004583991  0.0030718315
## [4,] 0.007459419 0.003928789  0.0068377132
## [5,] 0.007136834 0.005641238  0.0077328617
## [6,] 0.007424232 0.002128582  0.0070296959
```

and the loadings matrix $\mathbf{\Lambda}$ through

```
pca$rotation
```

```
##              PC1          PC2         PC3         PC4         PC5          PC6
## X3M  -0.2497417  0.408937681 -0.46710638 -0.12863509  0.27300216  0.41792512
## X6M  -0.2506220  0.408419141 -0.31399895  0.07040987  0.16598970 -0.03267909
## X1Y  -0.2535524  0.367509360 -0.05516733  0.22836416 -0.13001112 -0.51227975
## X2Y  -0.2583524  0.259830393  0.19435716  0.06882252 -0.31901598 -0.31344496
## X3Y  -0.2610532  0.167557746  0.25749140 -0.05798677 -0.35516360  0.15908084
## X4Y  -0.2625452  0.079431266  0.28869369 -0.05126628 -0.24587403  0.36471771
## X5Y  -0.2630606  0.007406988  0.28195606 -0.09568414 -0.01071591  0.16631008
## X6Y  -0.2629331 -0.056046922  0.25295080 -0.06084063  0.15156495  0.13236685
## X7Y  -0.2624446 -0.109559002  0.19138781 -0.03878492  0.25909045  0.11661320
## X8Y  -0.2617655 -0.151612235  0.12646471 -0.04484831  0.31428737 -0.01264216
```

```
## X9Y  -0.2611098 -0.180839353  0.06830235 -0.08265683  0.33838014 -0.24766803
## X10Y -0.2605678 -0.201515720  0.01242559 -0.05116883  0.25586249 -0.35221412
## X15Y -0.2553810 -0.323758677 -0.19272684  0.54903838 -0.04345030  0.08141030
## X20Y -0.2540945 -0.345000065 -0.29960276  0.37660799 -0.28957137  0.17864329
## X30Y -0.2551738 -0.306042715 -0.40610502 -0.66822020 -0.37141691 -0.14964249
##              PC7         PC8         PC9        PC10        PC11         PC12
## X3M   0.37192413 -0.13777806  0.31126733  0.16544523  0.04234539 -0.042623606
## X6M  -0.36608592  0.19624417 -0.51289633 -0.37823082 -0.23181915  0.064417155
## X1Y  -0.28375814 -0.09882866  0.11777700  0.49034476  0.34467451 -0.007550128
## X2Y   0.28170854  0.11215793  0.40851389 -0.42939859 -0.32140688 -0.184063449
## X3Y   0.28265796 -0.08044107 -0.18467666 -0.12475131  0.23949213  0.526045241
## X4Y   0.01269644  0.02652855 -0.37421346  0.07230895  0.31627457 -0.469380363
## X5Y  -0.16492181 -0.12428524 -0.01534060  0.38533484 -0.61706741 -0.244315083
## X6Y  -0.24248146  0.12141430  0.19337779  0.14703160 -0.12622712  0.393780894
## X7Y  -0.22760208  0.13651071  0.22409901 -0.11063217  0.12697854  0.290897992
## X8Y  -0.14018760  0.06108339  0.19619112 -0.26340913  0.28616560 -0.285053061
## X9Y   0.13790553 -0.21923032 -0.07910644 -0.16825912  0.17687398 -0.216586932
## X10Y  0.42979751 -0.22162269 -0.37713539  0.16370367 -0.18504057  0.178561342
## X15Y  0.24768955  0.59878300 -0.03141823  0.18529581 -0.01475529 -0.039550127
## X20Y -0.22976303 -0.59488354  0.09002347 -0.19742935 -0.06219228  0.044554108
## X30Y -0.11127310  0.22544120  0.03509861  0.06171501  0.02545533 -0.009072266
##             PC13          PC14         PC15
## X3M   0.03509922 -7.634855e-03 -0.001461526
## X6M  -0.03036490 -2.417579e-02  0.012687508
## X1Y  -0.02489493  4.433380e-02 -0.009112577
## X2Y   0.20395433 -9.368141e-03 -0.033753331
## X3Y  -0.45473027  2.128001e-02  0.072912657
## X4Y   0.41869054 -2.354515e-02 -0.054611761
## X5Y  -0.40058997  1.307909e-01 -0.070076815
## X6Y   0.32904544 -5.633123e-01  0.300675895
## X7Y   0.17120810  4.713665e-01 -0.561353770
## X8Y  -0.25355267  2.629973e-01  0.603067045
## X9Y  -0.26525440 -5.349217e-01 -0.418851529
## X10Y  0.33402335  2.844289e-01  0.194254245
## X15Y -0.14289622 -5.866589e-02 -0.044751811
## X20Y  0.09313888  5.744344e-03  0.027012865
## X30Y -0.01332056  4.170108e-05 -0.016954656
```

Step 3: Data approximation with a lower dimensional set of principal components. Returning to the representation in equation 12, we already noted that the principal components contained in $\mathbf{X}$ are sorted by their contribution to explaining variation in the original (standardized) data matrix $\mathbf{D}$ – the leftmost column of $\mathbf{X}$ contains the first principal component which explains the most variance, and the rightmost column contains the last principal component which captures the least variance. To reduce dimensionality of the original data while retaining as much information as possible, we can therefore only keep the first $K^* < K$ columns of $\mathbf{X}$ and throw away the principal components contained in columns after column K^*. Combining the first K^* principal components with their

corresponding elements in the loadings matrix $\mathbf{\Lambda}$, the approximation $\mathbf{D}^*$ of the original data then is:

$$\underbrace{\mathbf{D}^*}_{(N\times K)} = \underbrace{\mathbf{X}^*}_{(N\times K^*)}\underbrace{\mathbf{\Lambda}'^*}_{(K^*\times K)} \tag{14}$$

For example, if we choose to keep only the first two principal components ($K^* = 2$), equation 14 becomes

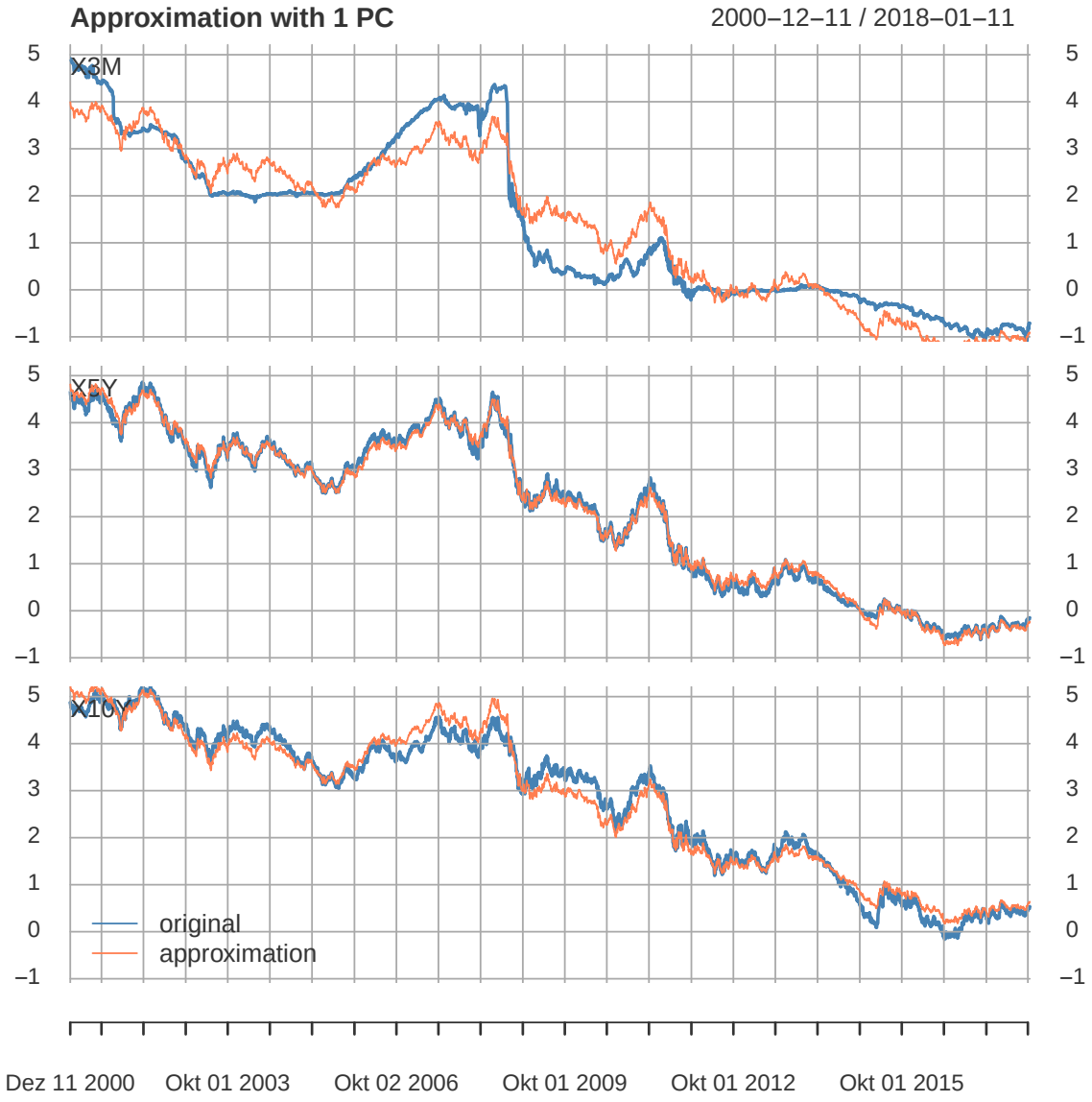
$$\underbrace{\begin{pmatrix} d_1^{*3M} & d_1^{*6M} & d_1^{*1Y} & \dots & d_1^{*30Y} \\ d_2^{*3M} & d_2^{*6M} & d_2^{*1Y} & \dots & d_2^{*30Y} \\ \vdots & \vdots & \vdots & & \vdots \\ d_N^{*3M} & d_N^{*6M} & d_N^{*1Y} & \dots & d_N^{*30Y} \end{pmatrix}}_{\mathbf{D}^*(N\times K)} = \underbrace{\begin{pmatrix} pc_1^1 & pc_1^2 \\ pc_2^1 & pc_2^2 \\ \vdots & \vdots \\ pc_N^1 & pc_N^2 \end{pmatrix}}_{\mathbf{X}(N\times K^*)} \underbrace{\begin{pmatrix} \lambda'_{1,1} & \lambda'_{1,2} & \lambda'_{1,3} & \dots & \lambda'_{1,15} \\ \lambda'_{2,1} & \lambda'_{2,2} & \lambda'_{2,3} & \dots & \lambda'_{2,15} \end{pmatrix}}_{\mathbf{\Lambda}'(K^*\times K)}$$

Let's check how well we can approximate our original yield data using just the first principal component. For the purpose of illustration, we will visually check results for just the 3M, the 5Y and the 10Y maturities. Note that to compare our approximation $\mathbf{D}^*$ to the original data, we have do revert the standardization by multiplying by the standard deviation of the respective maturity and adding back the mean.

```
# get mean and SD of original data:
yields_mean <- colMeans(yields[,-1]) #drop 1st col with dates
yields_sd <- apply(yields[,-1], 2, sd)

# D* = X* Lambda'*
#note that we need to transpose Lambda (pca$rotation)!
# drop=FALSE: 1 column of matrix retains matrix dimensions
# instead of automatically becoming just a vector
yields_approx_1pc <- pca$x[,1, drop=FALSE] %*% t(pca$rotation)[1,]
# scale to original SD and add back the mean
for(i in 1:ncol(yields_approx_1pc)){
  yields_approx_1pc[,i] <- yields_approx_1pc[,i] * yields_sd[i] + yields_mean[i]
}


colnames(yields_approx_1pc) <- colnames(yields[,-1])
yields_approx_1pc.xts <- xts(yields_approx_1pc, order.by=time(yields.xts))
head(yields_approx_1pc.xts)
```

```
##                 X3M      X6M      X1Y      X2Y      X3Y      X4Y      X5Y
## 2000-12-11 3.983339 4.027761 4.089700 4.254092 4.469743 4.676231 4.806402
## 2000-12-12 3.991593 4.035981 4.097918 4.262445 4.478295 4.684854 4.814904
## 2000-12-13 3.942626 3.987218 4.049161 4.212891 4.427559 4.633704 4.764471
## 2000-12-14 3.908776 3.953510 4.015457 4.178636 4.392488 4.598346 4.729608
## 2000-12-15 3.873499 3.918380 3.980332 4.142936 4.355937 4.561497 4.693276
## 2000-12-18 3.850012 3.894992 3.956946 4.119168 4.331603 4.536964 4.669086
##                 X6Y      X7Y      X8Y      X9Y     X10Y     X15Y     X20Y
## 2000-12-11 4.950731 5.079217 5.174430 5.207283 5.242642 5.594566 5.728898
```

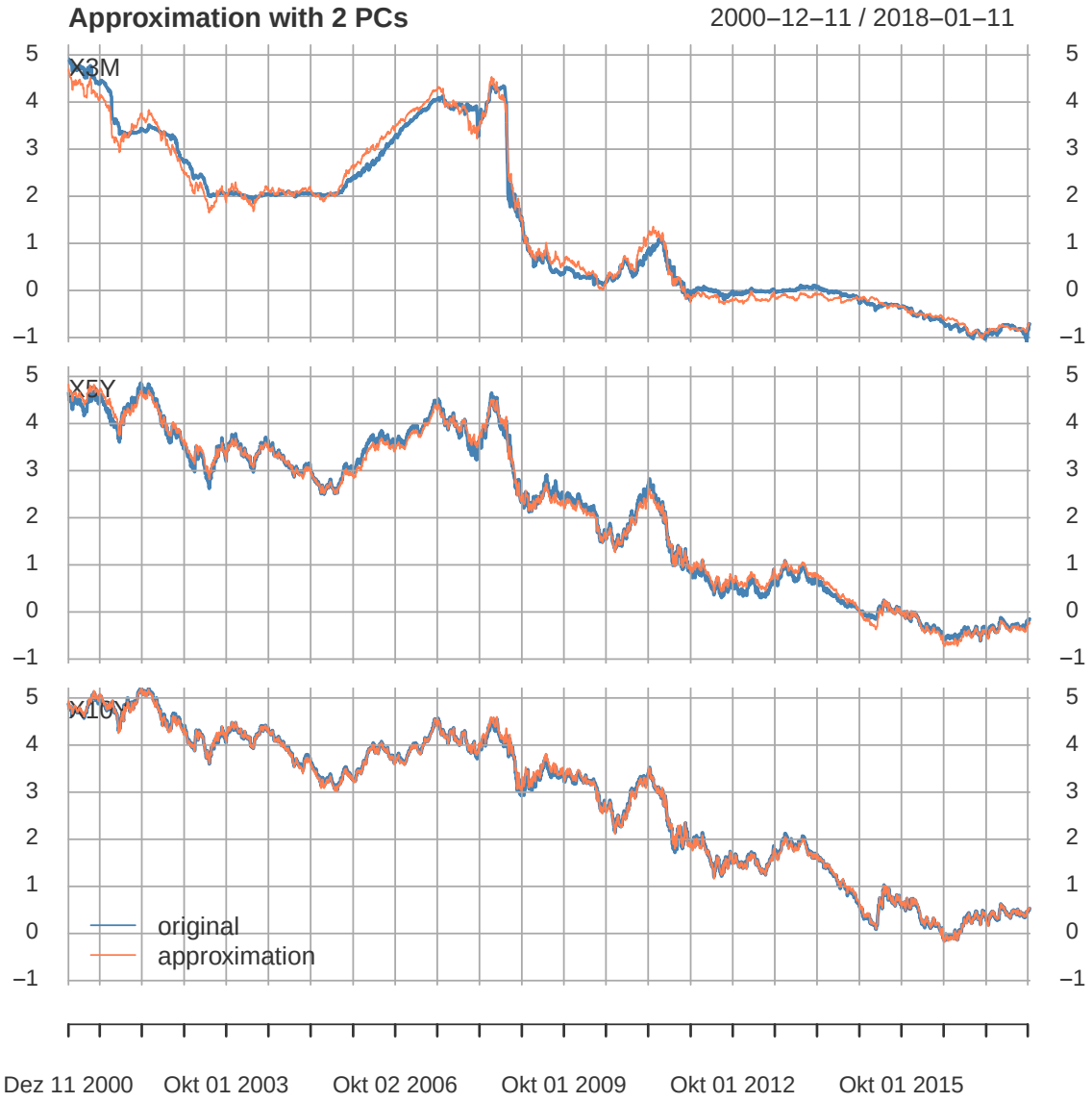
```
## 2000-12-12 4.959179 5.087586 5.182660 5.215267 5.250415 5.602008 5.736198
## 2000-12-13 4.909060 5.037934 5.133838 5.167900 5.204304 5.557862 5.692893
## 2000-12-14 4.874414 5.003611 5.100088 5.135157 5.172429 5.527345 5.662958
## 2000-12-15 4.838308 4.967840 5.064915 5.101032 5.139210 5.495541 5.631760
## 2000-12-18 4.814269 4.944025 5.041498 5.078313 5.117094 5.474367 5.610989
##                X30Y
## 2000-12-11 5.676231
## 2000-12-12 5.683355
## 2000-12-13 5.641094
## 2000-12-14 5.611880
## 2000-12-15 5.581434
## 2000-12-18 5.561164

plot_1pc <- plot(yields.xts[,c("X3M", "X5Y", "X10Y")],
               col="steelblue", multi.panel=TRUE, main="Approximation with 1 PC")
plot_1pc <- lines(yields_approx_1pc.xts[,"X3M"],
                col="coral", on=1)
plot_1pc <- lines(yields_approx_1pc.xts[,"X5Y"],
                col="coral", on=2)
plot_1pc <- lines(yields_approx_1pc.xts[,"X10Y"],
                col="coral", on=3)
plot_1pc <- addLegend("bottomleft", legend.names=c("original", "approximation"),
                    col=c("steelblue", "coral"), lwd=1, on=3)
plot_1pc
```



We see that using just the first PC the data is already described pretty well. How much does our approximation improve if we take the first two principal components instead?

```
# D* = X* Lambda'*
#note that we need to transpose Lambda (pca$rotation)!
yields_approx_2pc <- pca$x[,1:2] %*% t(pca$rotation)[1:2,]
# scale to original SD and add back the mean
for(i in 1:ncol(yields_approx_2pc)){
  yields_approx_2pc[,i] <- yields_approx_2pc[,i] * yields_sd[i] + yields_mean[i]
}


colnames(yields_approx_2pc) <- colnames(yields[,-1])
yields_approx_2pc.xts <- xts(yields_approx_2pc, order.by=time(yields.xts))

plot_2pc <- plot(yields.xts[,c("X3M", "X5Y", "X10Y")],
                 col="steelblue", multi.panel=TRUE, main="Approximation with 2 PCs")
```

```
plot_2pc <- lines(yields_approx_2pc.xts[,"X3M"],
                  col="coral", on=1)
plot_2pc <- lines(yields_approx_2pc.xts[,"X5Y"],
                  col="coral", on=2)
plot_2pc <- lines(yields_approx_2pc.xts[,"X10Y"],
                  col="coral", on=3)
plot_2pc <- addLegend("bottomleft", legend.names=c("original", "approximation"),
                      col=c("steelblue", "coral"), lwd=1, on=3)
plot_2pc
```



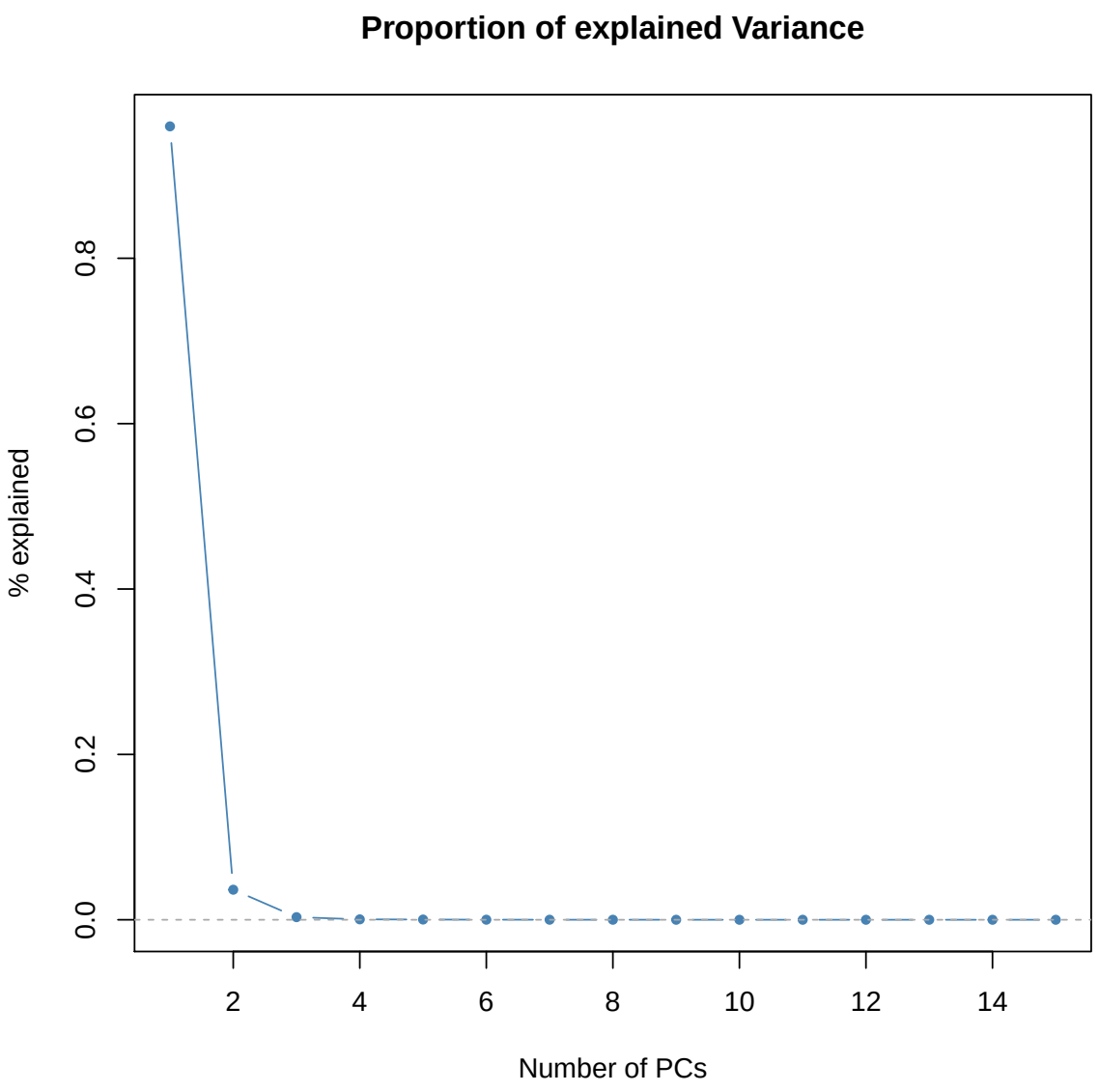
For all three maturities, but especially for the 3M and the 10Y, we observe that the approximation considerably improves. Adding the third, fourth,... principal component would further improve the fit to the original data – taking all 15 principal components, our approximation would in fact perfectly match the original data, but there would be no dimension reduction any more. How can we choose the "optimal" number of principal components, i.e. retain sufficient information from the raw data while keeping the number of principal components as small as possible? The variation in all principal components

together exactly describes the variation in the original data, so we can compute the fraction of variation in the original data captured by principal component i as

$$\frac{Var(PC_i)}{\sum_{k=1}^{K} Var(PC_k)}.$$

Plotting proportions of the overall variance attributable to individual principal components can help reveal whether adding another principal component captures meaningful additional variation in the data. For this purpose, `prcomp()` provides standard deviations of principal components, which can be directly converted to variances and further to proportions:

```
plot(pca$sdev^2/sum(pca$sdev^2),
     col="steelblue", type="b", pch=20,
     main="Proportion of explained Variance", ylab="% explained"
     ,xlab="Number of PCs")
abline(h=0, lty=2, col="darkgrey")
```



As we can see, most of the variance can already be captured by the fist principal component alone. While for the second principal component, the proportion of variance explained is still somewhat meaningful, for the third principal component, it is already just above the zero line. For all further principal components, the fraction of variance explained is indistinguishable from zero in our plot. Depending of the desired degree of precision, between 1 and 3 principal components to approximate the original data therefore appear reasonable choice. Retaining any further principal components increases complexity, but adds virtually no additional information.

Step 4: Interpretation of principal components. Principal components are not automatically associated with a particular type of information, i.e., it is not always clear what type of information is captured by a particular principal component. Each principal component just captures the maximum amount of variation possible (and not yet captured by preceding principal components). However, in many cases, the data scientist can assign a meaningful interpretation to individual principal components by examining the loadings matrix $\mathbf{\Lambda}$, which governs how principal components relate to the raw data – recall equation 12:

$$\mathbf{D} = \mathbf{X}\mathbf{\Lambda}'$$

In the following, we focus on interpreting the variation of the first three principal components, which we just found to capture at least some meaningful fraction of the overall variation in the data. Let's have a look at their corresponding columns in the loadings matrix $\mathbf{\Lambda}$ (note: to arrive at the representation in 12, we would have to transpose the matrix):

```
pca$rotation[,1:3]
```

```
##              PC1          PC2         PC3
## X3M  -0.2497417  0.408937681 -0.46710638
## X6M  -0.2506220  0.408419141 -0.31399895
## X1Y  -0.2535524  0.367509360 -0.05516733
## X2Y  -0.2583524  0.259830393  0.19435716
## X3Y  -0.2610532  0.167557746  0.25749140
## X4Y  -0.2625452  0.079431266  0.28869369
## X5Y  -0.2630606  0.007406988  0.28195606
## X6Y  -0.2629331 -0.056046922  0.25295080
## X7Y  -0.2624446 -0.109559002  0.19138781
## X8Y  -0.2617655 -0.151612235  0.12646471
## X9Y  -0.2611098 -0.180839353  0.06830235
## X10Y -0.2605678 -0.201515720  0.01242559
## X15Y -0.2553810 -0.323758677 -0.19272684
## X20Y -0.2540945 -0.345000065 -0.29960276
## X30Y -0.2551738 -0.306042715 -0.40610502
```

We observe that the first principal component is almost uniformly related to yields of all maturities: loadings in the first column are all extremely similar in magnitude. So if the principal component goes up by one standard deviation, all yields go down by roughly the same amount. We can therefore interpret the first principal component as capturing the *level* of the yield curve. Note that principal components and loadings

are only defined up to sign – we could multiply the first principal component and every entry in the first column of $\mathbf{\Lambda}$ with (-1) to arrive at the same outcome in equation 12. For the second principal component, loadings in column two flip sign from the short to the long end of the yield curve. As a result, if the second principal component goes up by one standard deviation, the short end of the yield curve goes up but yields at long maturities decrease, i.e. the yield curve gets flatter. Accordingly, we can interpret the second principal component as capturing the *slope* of the yield curve. Finally, the loadings in the third column are negative for both short and long maturities, but positive for medium-term maturities – if the third principal component goes up by one standard deviation, yields at both the short and the long end decrease, but increase in the middle of the yield curve. We therefore interpret the third principal component as capturing the *curvature* of the yield curve.

Appendix

A. Modern Portfolio Theory

A.1. Simplified Model

Consider the following one period model.

- There are two stocks, A and B, with normally distributed returns, r_A and r_B.
- The stocks are perfectly divisible, i.e. it is possible to hold, e.g., a 10^{th} of a share.
- There are no transaction costs or taxes. If you buy stock A you only pay its price.
- Investors' utility is only a function of expected return and volatility.
- Investors are risk-averse.

The goal of an investor is it to find the portfolio with minimum variance given an expected return target. The portfolio return is given by

$$r_P = w_A r_A + w_B r_B, \tag{15}$$

where w_A and w_B are the portfolio weights of stock A and stock B. Since they are portfolios weights we have that

$$wA + w_b = 1. \tag{16}$$

The expected portoflio return is then given by the expectation of the portfolio return (15),

$$\mathbb{E}[r_P] = w_A \mathbb{E}[r_A] + w_B \mathbb{E}[r_B].$$

In the following we will abbreviate expected returns with μ, so the expected portfolio return is,

$$\mu_P = w_A \mu_A + w_B \mu_B. \tag{17}$$

The portfolio variance is given by

$$\begin{aligned}
\sigma_P^2 &= \mathbb{E}[(r_P - \mu_P)^2] = \mathbb{E}[r_P^2] - 2\mu_P \mathbb{E}[r_P] + \mu_P^2 = \mathbb{E}[r_P^2] - \mu_P^2 \\
\text{substituting for } r_P \quad &= \mathbb{E}[(w_A r_A + w_B r_B)^2] - (w_A \mu_A + w_B \mu_B)^2 \\
&= w_A^2 \left(\mathbb{E}[r_A^2] - \mu_A^2\right) + w_B^2 \left(\mathbb{E}[r_B^2] - \mu_B^2\right) + 2 w_A w_B \left(\mathbb{E}[r_A r_B] - \mu_A \mu_B\right)
\end{aligned}$$

Recall that $\rho_{AB} = \frac{\text{Cov}(r_A, r_B)}{\sigma_A \sigma_B}$,

$$\sigma_P^2 = w_A^2 \sigma_A^2 + w_B^2 \sigma_B^2 + 2 w_A w_B \rho_{AB} \sigma_A \sigma_B, \tag{18}$$

$$\sigma_P = \sqrt{w_A^2 \sigma_A^2 + w_B^2 \sigma_B^2 + 2 w_A w_B \rho_{AB} \sigma_A \sigma_B}. \tag{19}$$

Now, we have found function that return the expected portfolio return, (17) and the portfolio volatility, (19), if we input w_A and w_B. Note, that since there is a unique

combination of w_A and w_B for each expected portfolio return the corrspoding portfolio variance is the minimum variance for that expected return.

But what we really want to find is function that output, w_A, w_B and σ_P when we input μ_P. Observe that

$$\begin{aligned}
\mu_P &= w_A\mu_A + w_B\mu_B w_A \\
\text{using } w_B = 1 - w_A \text{ we get} \quad &= \mu_A + (1-w_A)\mu_B = w_A(\mu_A - \mu_B) + \mu_B \\
\mu_P - \mu_B &= w_A(\mu_A - \mu_B) \\
\implies w_A &= \frac{\mu_P - \mu_B}{\mu_A - \mu_B}, \qquad (20) \\
w_B &= 1 - w_A = 1 - \frac{\mu_P - \mu_B}{\mu_A - \mu_B} = -\frac{\mu_P - \mu_A}{\mu_A - \mu_B}. \qquad (21)
\end{aligned}$$

So we found function for w_A and w_B. We can use these to also find a function that outputs σ_P when we input μ_P:

$$\begin{aligned}
\sigma_P^2 &= w_A^2\sigma_A^2 + w_B^2\sigma_B^2 + 2w_A w_B \rho_{AB}\sigma_A\sigma_B \\
&= \left(\frac{\mu_P - \mu_B}{\mu_B - \mu_A}\right)^2 \sigma_A^2 + \left(-\frac{\mu_P - \mu_A}{\mu_B - \mu_A}\right)^2 \sigma_B^2 - 2\left(\frac{(\mu_P - \mu_B)(\mu_P - \mu_A)}{(\mu_B - \mu_A)^2}\right)\rho_{AB}\sigma_A\sigma_B, \\
\sigma_P &= \sqrt{\frac{1}{((\mu_B - \mu_A)^2)}\left((\mu_P - \mu_B)^2\sigma_A^2 + (\mu_P - \mu_A)^2\sigma_B^2 - 2(\mu_P - \mu_B)(\mu_P - \mu_A)\rho_{AB}\sigma_A\sigma_B\right)} \qquad (22)
\end{aligned}$$

A.2. Multiple Assets Model

Consider the following one period model.

- There are n assets with normally distributed returns, $r_i, i = 1, \ldots, n$.
- The assets are perfectly divisible, i.e. it is possible to hold, e.g., a 10^{th} of a share.
- Investors are risk-averse.
- Investors' utility is only a function of expected return and volatility.
- There are no transaction costs or taxes. If you buy stock A you only pay its price.
- No asset is a perfect substitute for another asset.

Let us start with some notation: $r = (r_1, \ldots, r_n)'$ denotes the vector of asset returns, $\mu = (\mu_1, \ldots, \mu_n)'$ denotes the vector of expected asset returns, $w = (w_1, \ldots, w_n)'$ denotes the vector of portfolio weights and Σ denotes the variance-covariance matrix. Σ has the following form

$$\Sigma = \begin{pmatrix} \sigma_1^2 & \ldots & \sigma_{1n} \\ \vdots & \ddots & \\ \sigma_{n1} & \ldots & \sigma_n^2 \end{pmatrix},$$

here σ_i^2 represents asset i's return variance and σ_{ij} the covariance of returns between assets i and j for $i \neq j$. Since we assumed that no asset is a perfect substitute for another asset, i.e. assets are not perfectly correlated, we have that Σ is positive definite and thus invertible.

Recall from above that the expected portfolio return is given by the sum of expected asset return weighted by the portfolio weight. So,

$$\mu_P = \mu' w, \tag{23}$$

and the portfolio variance is given by the sum of all variances weighted by their sqaured portfolio weight plus the sum of covariances times their respective weights times two, or simply

$$\sigma_P^2 = w' \Sigma w. \tag{24}$$

The problem a portfolio manager has is to find the portfolio that minimizes variances given an expected return target.

So, $\min \sigma_P^2$ given that $\mu' w = \mu_P$. Note, that we also have the technical constraint that all portfolio weights need to sum to one, i.e. $\mathbf{1}' w = 1$, where $\mathbf{1}$ is a n-dimensional vector full of ones. We will now write down the full optimization problem that the portfolio manager needs to solve:

$$\min_w \frac{1}{2} w' \Sigma w \qquad \text{s.t. } \mu' w = \mu_P, \quad \mathbf{1}' w = 1 \tag{25}$$

Note, that we added $\frac{1}{2}$ at the very beginning. This is only for computational convenience as you will see in a second. To solve this problem we need to write down the corresponding Lagrange function, $\mathcal{L}$:

$$\mathcal{L} = \frac{1}{2} w' \Sigma w - \lambda_1 (\mu' w - \mu_P) - \lambda_2 (\mathbf{1}' w - 1).$$

Let us now differentiate with respect to w and set the first derivative to zero to get the first order condition (FOC)

$$\frac{\partial \mathcal{L}}{\partial w} = w' \Sigma - \lambda_1 \mu' - \lambda_2 \mathbf{1}' = 0 \tag{26}$$

By the FOC (26) we have that the optimal portfolio weights, $w*$, or in other words, the weights minimizing the portfolio variance given the expected return target μ_P, are,

$$w^{*'} = (\lambda_1 \mu' + \lambda_2 \mathbf{1}') \, \Sigma^{-1}. \tag{27}$$

Nice, but we still needs to find the Lagrange multipliers λ_1 and λ_2, because otherwise this expression is pretty useless to us. To find the multipliers we go back to the constraints and check them for $w*$, i.e. we substitue $w*$ for w. Remember the constraints are $\mu' w = \mu_P = w' \mu$ and $\mathbf{1}' w = 1 = w' \mathbf{1}$.

$$
\begin{aligned}
w^{*'}\mu &= \left(\left(\lambda_1\mu' + \lambda_2\mathbf{1}'\right)\Sigma^{-1}\right)\mu = \left(\lambda_1\mu'\Sigma^{-1} + \lambda_2\mathbf{1}'\Sigma^{-1}\right)\mu \\
&= \lambda_1\mu'\Sigma^{-1}\mu + \lambda_2\mathbf{1}'\Sigma^{-1}\mu \\
\text{and since } \mu_P = w'\mu \implies \mu_P &= \lambda_1\mu'\Sigma^{-1}\mu + \lambda_2\mathbf{1}'\Sigma^{-1}\mu, \\
w^{*'}\mathbf{1} &= \left(\left(\lambda_1\mu' + \lambda_2\mathbf{1}'\right)\Sigma^{-1}\right)\mathbf{1} = \left(\lambda_1\mu'\Sigma^{-1} + \lambda_2\mathbf{1}'\Sigma^{-1}\right)\mathbf{1} \\
&= \lambda_1\mu'\Sigma^{-1}\mathbf{1} + \lambda_2\mathbf{1}'\Sigma^{-1}\mathbf{1} \\
\text{and since } 1 = w'\mathbf{1} \implies 1 &= \lambda_1\mu'\Sigma^{-1}\mathbf{1} + \lambda_2\mathbf{1}'\Sigma^{-1}\mathbf{1}.
\end{aligned}
$$

Let $\alpha := \mu'\Sigma^{-1}\mu, \beta := \mathbf{1}'\Sigma^{-1}\mu$ and $\gamma := \mathbf{1}'\Sigma^{-1}\mathbf{1}$ and substitute above to get

$$
\begin{aligned}
\mu_P &= \lambda_1\mu'\Sigma^{-1}\mu + \lambda_2\mathbf{1}'\Sigma^{-1}\mu = \lambda_1\alpha + \lambda_2\beta, \\
1 &= \lambda_1\mu'\Sigma^{-1}\mathbf{1} + \lambda_2\mathbf{1}'\Sigma^{-1}\mathbf{1} \\
&\text{As } \Sigma \text{ is symmetric and } \beta \text{ is a scalar we have that} \\
\beta &= \beta' = \left(\mathbf{1}'\Sigma^{-1}\mu\right)' \\
&= \mu'\left(\mathbf{1}'\Sigma^{-1}\right)' = \mu'\left(\Sigma^{-1}\right)'\mathbf{1} = \mu'\Sigma^{-1}\mathbf{1}. \\
\text{Hence,} \quad \beta &= \mathbf{1}'\Sigma^{-1}\mu = \mu'\Sigma^{-1}\mathbf{1}. \\
1 &= \lambda_1\mu'\Sigma^{-1}\mathbf{1} + \lambda_2\mathbf{1}'\Sigma^{-1}\mathbf{1} = \lambda_1\beta + \lambda_2\gamma.
\end{aligned}
$$

Now, we can solve for the Lagrange multipliers

$$
\begin{aligned}
\lambda_2 = \frac{1-\lambda_1\beta}{\gamma} &\implies \mu_P = \lambda_1\alpha + \frac{1-\lambda_1\beta}{\gamma}\beta \\
&= \frac{\lambda_1\alpha\gamma + \beta - \lambda_1\beta^2}{\gamma} = \frac{\lambda_1\left(\alpha\gamma - \beta^2\right) + \beta}{\gamma}, \qquad (28) \\
\implies \lambda_1 &= \frac{\gamma\mu_P - \beta}{(\alpha\gamma - \beta^2)}, \qquad (29) \\
\lambda_2 &= \frac{\alpha - \beta\mu_P}{(\alpha\gamma - \beta^2)}. \qquad (30)
\end{aligned}
$$

Now, we can use (27) to find the optimal portfolio weights corresponding to the desired expected return μ_P. We still want to find the minimum portfolio variance σ_P^2 given an expected return target. We can rearrange (26) and get

$$
w'\Sigma = \lambda_1\mu' + \lambda_2\mathbf{1}'
$$

If we multiply this with w from the right, we get an expression for the portfolio variance as defined in (24):

$$
\begin{aligned}
w'\Sigma w &= \lambda_1 \mu' w + \lambda_2 \mathbf{1}' w \\
\implies \sigma_P^2 &= \frac{\gamma\mu_P - \beta}{(\alpha\gamma - \beta^2)}\mu' w + \frac{\alpha - \beta\mu_P}{(\alpha\gamma - \beta^2)}\mathbf{1}' w \\
&= \frac{\gamma\mu_P \mu' w - \beta\mu' w + \alpha \mathbf{1}' w - \beta\mu_P \mathbf{1}' w}{(\alpha\gamma - \beta^2)} \\
&= \frac{\gamma\mu_P \overbrace{\mu' w}^{=\mu_P} - \beta \overbrace{\mu' w}^{=\mu_P} + \alpha \overbrace{\mathbf{1}' w}^{=1} - \beta\mu_P \overbrace{\mathbf{1}' w}^{=1}}{(\alpha\gamma - \beta^2)} \\
&= \frac{\gamma\mu_P^2 - \beta\mu_P + \alpha - \beta\mu_P}{(\alpha\gamma - \beta^2)} = \frac{\gamma\mu_P^2 - 2\beta\mu_P + \alpha}{(\alpha\gamma - \beta^2)}
\end{aligned} \tag{31}
$$

Now, we have found functions that output the optimal portfolio weights, (27), and the corresponding (minimum) variance, (31). Lastly, if we want to find the portfolio that has the minimum variance of all possible portfolios we need to differentiate (28) and (31) with respect to μ_P and set the results to zero. The first differentiating will yield to expected return of that portfolio and the second one the variance. You can try this yourselves, the results should be:

$$
\begin{aligned}
\mu_P &= \frac{\beta}{\gamma}, \\
\sigma_P^2 &= \frac{1}{\gamma}.
\end{aligned}
$$

B. OLS Estimates

B.1. Univariate OLS

$$\begin{aligned}
\min f(a,b) = \min \sum_{i=1}^{N} e_i^2 &= \sum_{i=1}^{N} (y_i - a - bx_i)^2 = \sum_{i=1}^{N} \left(y_i^2 + 2y_i(-a-bx_i) + (-a-bx_i)^2\right) \\
&= \sum_{i=1}^{N} \left(y_i^2 - 2y_i a - 2y_i b x_i + a^2 + 2abx_i + b^2 x_i^2\right) \\
\frac{\partial f(a,b)}{\partial a} &= \sum_{i=1}^{N} -2y_i + \sum_{i=1}^{N} 2a + \sum_{i=1}^{N} 2bx_i = 0 \\
2a\sum_{i=1}^{N} &= 2\sum_{i=1}^{N} y_i - 2b\sum_{i=1}^{N} x_i \\
aN = \sum_{i=1}^{N} y_i - b\sum_{i=1}^{N} x_i &\implies a = \frac{\sum_{i=1}^{N} y_i}{N} - b\frac{\sum_{i=1}^{N} x_i}{N} = \bar{y} - b\bar{x} \\
\frac{\partial f(a,b)}{\partial b} &= -2\sum_{i=1}^{N} y_i x_i + 2a\sum_{i=1}^{N} x_i + 2b\sum_{i=1}^{N} x_i^2 = 0 \\
-\sum_{i=1}^{N} y_i x_i + a\sum_{i=1}^{N} x_i + b\sum_{i=1}^{N} x_i^2 &= 0 \\
-\sum_{i=1}^{N} y_i x_i + (\bar{y} - b\bar{x})\sum_{i=1}^{N} x_i + b\sum_{i=1}^{N} x_i^2 &= 0
\end{aligned}$$

Now we use that $\sum_{i=1}^{N} x_i = N\bar{x}$

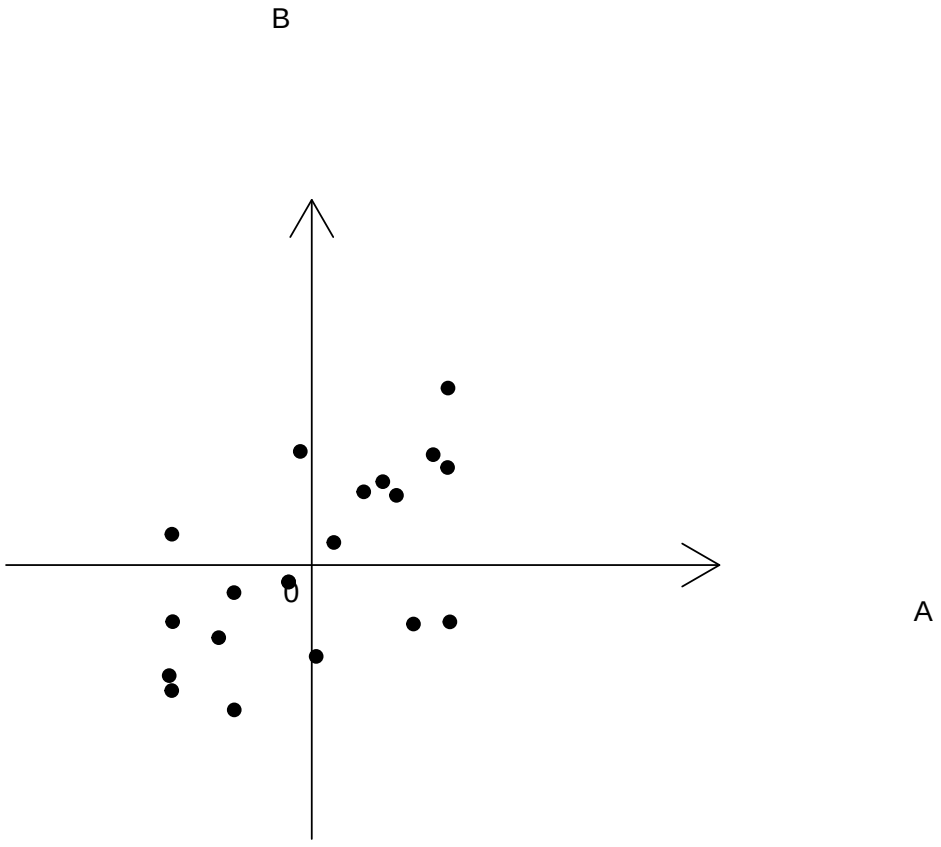
$$\begin{aligned}
-\sum_{i=1}^{N} y_i x_i + (\bar{y} - b\bar{x})\, N\bar{x} + b\sum_{i=1}^{N} x_i^2 &= 0 \\
-\sum_{i=1}^{N} y_i x_i + N\bar{x}\bar{y} + b\left(\sum_{i=1}^{N} x_i^2 - N\bar{x}^2\right) &= 0 \\
\implies b = \frac{\sum_{i=1}^{N} y_i x_i - N\bar{x}\bar{y}}{\sum_{i=1}^{N} x_i^2 - N\bar{x}^2} &= \frac{\mathrm{Cov}\,(x,y)}{\mathrm{Var}\,(x)}.
\end{aligned}$$

C. Principal Component Analysis - a geometric explanation

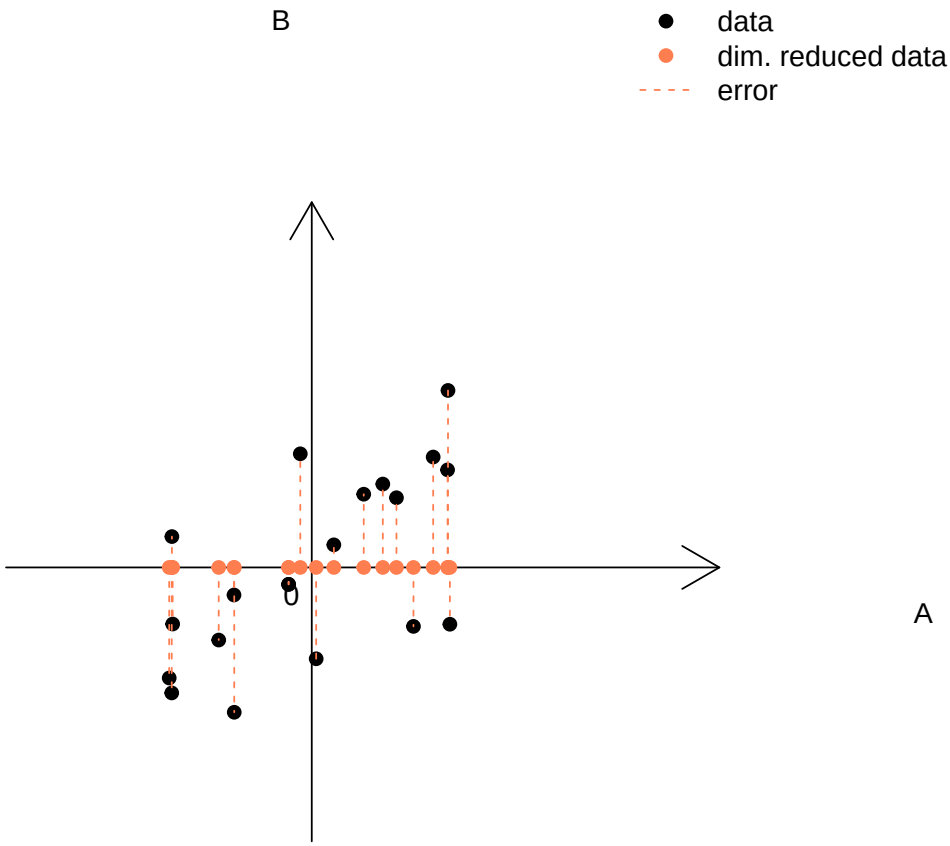
In this section we want to shed some more light on the intuition and the basic principles behind principal component analysis (PCA). However, the following should not be thought of as a formal derivation of why PCA works.

Consider the following two dimensional data set.

```
##             [,1]       [,2]
## [1,]  1.3553627 -0.6220452
## [2,] -1.3738495 -1.3744479
## [3,] -0.7605606 -1.5860791
## [4,]  0.5100667  0.8022916
## [5,]  0.2167518  0.2476217
## [6,]  0.6972102  0.9133799
```



A single data point is fully identified by the tuple (A, B), where A and B are the coordinates in the two dimensional plane. Now, suppose we want to reduce the dimensionality of the data set, i.e., we want to describe the data using a single variable instead of two. Of course, using a single variable also implies that we will not be able to describe the data perfectly, but often the benefit of reducing dimensionality outweighs the drawback of making small *estimation errors*. A very crude way of reducing dimensionality is simply dropping one dimension. Let us look at the case where we only use dimension A. Assume we call the the first column of the data set x_A and the second column x_B, then we take a look at the vector x_A to describe the data.



The orange points represent the dimensionality reduced data and the orange dotted lines represent the errors we make if we approximate the original data using only the A dimension. By summing all the (euclidean) distances between the original and the approximated data points we get the overall estimation error.

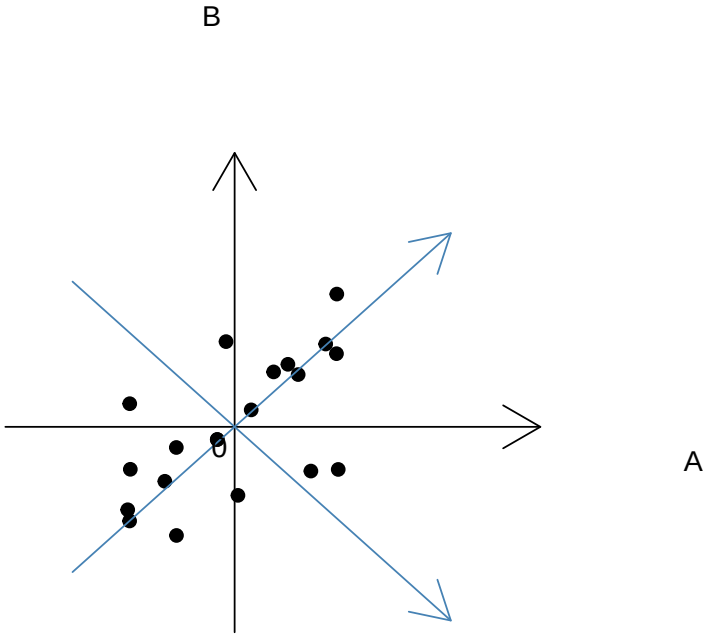
```
sum(sqrt(pcexdat[,2]^2))

## [1] 17.05235
```

PCA provides us with another way of reducing dimensionality. The goal of PCA is to find variables, the so called principal components, that explain as much of the variability in the data as possible. But how much variability is there? Before we continue be aware that we will use the terms varibility, variation and variance interchangeably in the following. The covariance matrix of the data provides the answer.

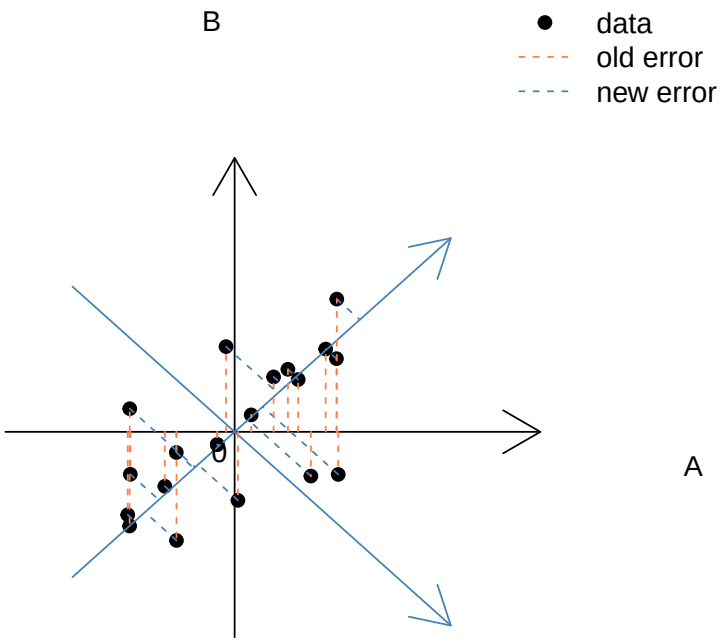
```
cov(pcexdat)

##           [,1]      [,2]
## [1,] 1.0000000 0.6173645
## [2,] 0.6173645 1.0000000
```

Along the A dimension, there is 1 *unit of variation*[17], or simply the variance of the vector x_A is 1. The variance of x_B is also 1, so the total variablility/variation/variance is 2. We started by using x_A as an approximation for the data (orange points in the plot above). How much variation in the data is explained by x_A? All of the A dimensional variation and we are left with all of the B dimensional variation. In order to find a variable that explains more variation we can use the covariance matrix of the data. Notice in the above example that by using x_A, which is equal to the "x-coordinate" in the plot, we explained the variation due to dimension A. If we pick x_B, then we explain the variation in dimension B or the "y-coordinate". When using PCA we try to rotate the coordinate system in such a way that one of the axes explains the maximal possible amount of variation. Basically, we change the basis of the vector space spanned by the covariance matrix such that the first "dimension" has maximal variation. Then we can "pick this dimension as variable" and end up with the variable that explains as much of the variation as possible of the given data set. This idea is represented visually by the blue lines, i.e. the axis of the "new coordinate system".

[17] This is because the data are standardized.



If we pick "the new x-axis" as variable the *estimation error* changes as following.




```
## [1] 8.234563
```

The new error is $\sim$ 8 and the error we make when using x_A is $\sim$ 17! This new axis is also called the first principal component. The second principal component is a variable that (in the two dimensional case) explains the remaining variation. Note that principal components do not explain the same variation. This means that the variable explaining the remaining variation is orthogonal to the first principal component. In the two dimensional case it is therefore immediately clear that the second variable is a vector in the direction of the second axis of the rotated coordinate system. It can be shown that the principal components of an $(N \times K)$-dimensional data set D are the K eigenvectors of the covariance matrix of D. Moreover, if the data are standardized properly and the eigenvectors have unit length, then each eigenvalue divided by the sum of all eigenvalues tells you how much of the variation the corresponding principal component explains.