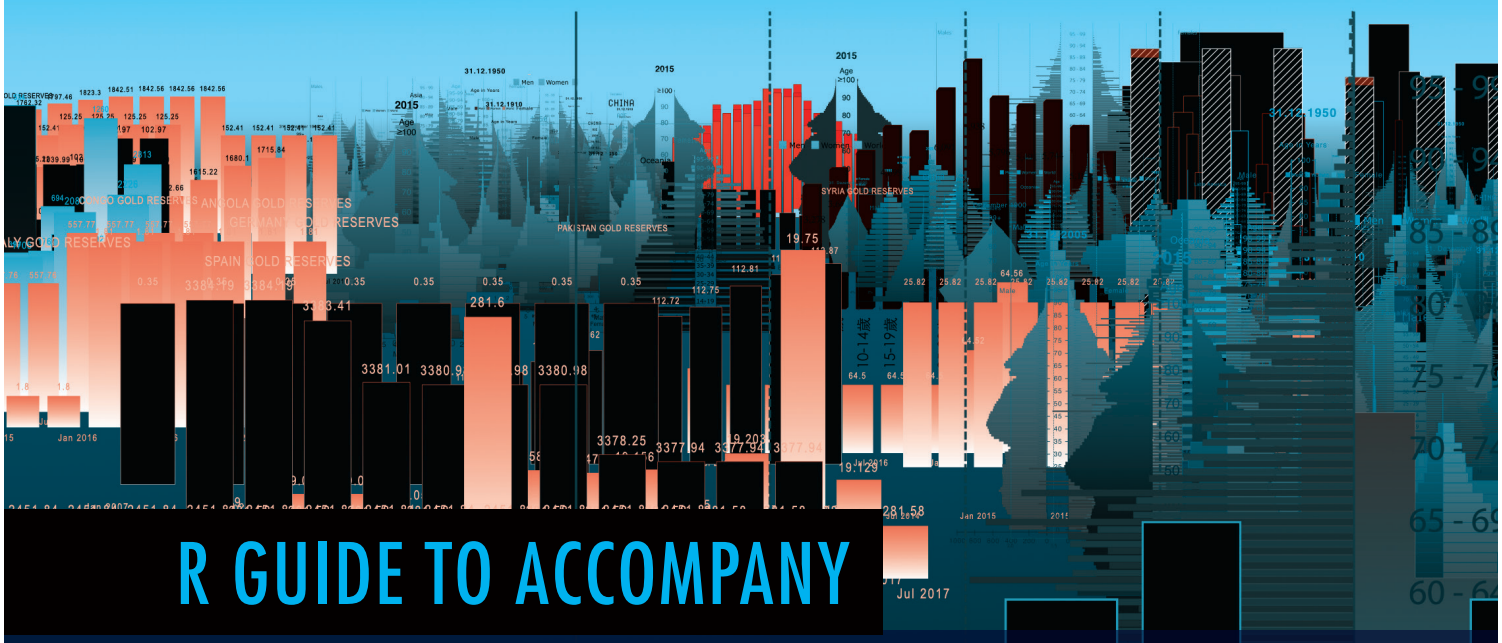
ROBERT WICHMANN
CHRIS BROOKS



R GUIDE TO ACCOMPANY

INTRODUCTORY ECONOMETRICS FOR FINANCE

4TH EDITION

This guide draws on material from 'Introductory Econometrics for Finance', published by Cambridge University Press, © Chris Brooks (2019). The Guide is intended to be used alongside the book, and page numbers from the book are given after each section and subsection heading.

The authors accept no responsibility for the persistence or accuracy of URLs for external or third-party internet websites referred to in this work, and nor do we guarantee that any content on such web sites is, or will remain, accurate or appropriate.

Contents

1 Introduction to R and RStudio **1**
1.1 What Are R and RStudio? . . . 1
1.2 What Does RStudio Look Like? . . . 1

2 Getting Started **3**
2.1 Packages . . . 3
2.2 Importing Data . . . 3
2.3 Data Description . . . 5
2.4 Changing and Creating Data . . . 6
2.5 Graphics and Plots . . . 7
2.6 Keeping Track of Your Work . . . 8

3 Linear Regression – Estimation of an Optimal Hedge Ratio **10**

4 Hypothesis Testing – Example 1: Hedging Revisited **13**

5 Hypothesis Testing – Example 2: The CAPM **15**

6 Sample Output for Multiple Hypothesis Tests **19**

7 Multiple Regression Using an APT-Style Model **20**
7.1 Stepwise Regression . . . 22

8 Quantile Regression **25**

9 Calculating Principal Components **28**

10 Diagnostic Testing **30**
10.1 Testing for Heteroscedasticity . . . 30
10.2 Using White's Modified Standard Error Estimates . . . 31
10.3 The Newey–West Procedure for Estimating Standard Errors . . . 32
10.4 Autocorrelation and Dynamic Models . . . 33
10.5 Testing for Non-Normality . . . 34
10.6 Dummy Variable Construction and Application . . . 35
10.7 Multicollinearity . . . 38
10.8 The RESET Test for Functional Form . . . 38
10.9 Stability Tests . . . 39
10.10 Recursive Estimation . . . 40

11 Constructing ARMA Models **43**
11.1 Estimating Autocorrelation Coefficients . . . 43
11.2 Using Information Criteria to Decide on Model Orders . . . 44

12 Forecasting Using ARMA Models **47**

13 Estimating Exponential Smoothing Models **49**

14 Simultaneous Equations Modelling **50**

15 Vector Autoregressive (VAR) Models **54**

16 Testing for Unit Roots **60**

17 Cointegration Tests and Modelling Cointegrated Systems **62**
17.1 The Johansen Test for Cointegration 64

18 Volatility Modelling **69**
18.1 Estimating GARCH Models 69
18.2 EGARCH and GJR Models 70
18.3 GARCH-M Estimation 72
18.4 Forecasting from GARCH Models 74
18.5 Estimation of Multivariate GARCH Models 75

19 Modelling Seasonality in Financial Data **78**
19.1 Dummy Variables for Seasonality 78
19.2 Estimating Markov Switching Models 79

20 Panel Data Models **82**

21 Limited Dependent Variable Models **87**

22 Simulation Methods **93**
22.1 Deriving Critical Values for a Dickey–Fuller Test Using Simulation 93
22.2 Pricing Asian Options 97

23 Value at Risk **100**
23.1 Extreme Value Theory 100
23.2 The Hill Estimator for Extreme Value Distributions 101
23.3 VaR Estimation Using Bootstrapping 102

24 The Fama–MacBeth Procedure **105**

References **108**

List of Figures

1 RStudio Main Windows . . . 2
2 Installing a Package in RStudio . . . 3
3 Importing Data . . . 4
4 The *Environment* Tab with Imported Data . . . 5
5 Line Plot of the Average House Price Series . . . 8
6 Histogram of Housing Return Series . . . 8
7 Time-series Plot of Two Series . . . 16
8 Generating a Scatter Plot of Two Series . . . 17
9 Plot of Coefficients from Quantile Regression Against OLS . . . 27
10 Plot of Residuals from Linear Regression . . . 30
11 Histogram of Residuals . . . 34
12 Regression Residuals and Fitted Series . . . 36
13 Plot of the Parameter Stability Test . . . 41
14 CUSUM Plot . . . 42
15 Autocorrelation and Partial Autocorrelation Functions . . . 43
16 Graph Comparing Static and Dynamic Forecasts with the Actual Series . . . 48
17 Graphs of Impulse Response Functions (IRFs) for the VAR(1) Model . . . 57
18 Graphs of FEVDs for the VAR(1) Model . . . 58
19 Graphs for FEVDs with Reverse Ordering . . . 59
20 Actual, Fitted and Residual Plot . . . 63
21 Graph of the Six US Treasury Interest Rates . . . 65
22 Graph of the Static and Dynamic Forecasts of the Conditional Variance . . . 75
23 State Probabilities Graph . . . 81
24 Graph of the Fitted Values from the Failure Probit Regression . . . 91
25 Changing the Way Code is Displayed in RStudio . . . 94
26 Hill Plot for Value at Risk . . . 102

List of Tables

1 Commonly Used Operators 7
2 Simulated Asian Option Prices 99
3 Fama–MacBeth Market Prices of Risk 107

1 Introduction to R and RStudio

1.1 What Are R and RStudio?

R is a language and environment for statistical computing and graphics. It provides a wide variety of statistical and graphical techniques, and is highly flexible. One of Rs strengths is the ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control.[1]

The most commonly used graphical integrated development environment for R is RStudio, which is also the one used in this guide.[2] A good way of familiarising yourself with RStudio is to work through the examples given in this guide. This section assumes that readers have installed RStudio and have successfully loaded it onto an available computer and that they have downloaded and installed a recent version of R (at the date of writing this guide, the latest version 3.5.2).

1.2 What Does RStudio Look Like?

When you start Rstudio you will be presented with a window which should resemble Figure 1. You will soon realise that the main window is actually subdivided into four smaller windows, headed by an application toolbar. The upper left *Source* window, which is minimised when starting a new session, will display R scripts or file once you open them, or you can browse through data within this window; you can also open several tabs like working on a script and having a look at a data set.

The lower left *Console* window is showing a prompt **'¿'** and can be used directly. It is the place where you can execute commands and where the output is displayed. A command is typed directly after the prompt and can be executed by pressing **Enter**. As such, you can use it as a calculator and for example type **1 + 1** into the *Console* and press **Enter**. You should obtain the result **[1] 2**, where the [1] only indicates the row of the result, followed by the actual result 2.

On the right-hand side of the screen, you will see two windows: the upper window with the tabs *Environment*, *History* and *Connections* and the lower window with the tabs *Files*, *Plots*, *Packages*, *Help* and *Viewer*. You can customise the location and number of tabs shown in RStudio by clicking **Tools/Global Options...** in the top toolbar, and then switch to the tab **Pane Layout** on the left of the pop up window. However, we will keep the default arrangement here.

We will not discuss in detail every one of these tabs, but instead focus on the main tabs that we will use in this guide. The *Environment* tab shows the data and variables that have been loaded and created in the current workspace as such, it is helpful to get a quick information on size, elements, formats and other characteristics of the data. The *History* tab acts as a log and can be helpful to find commands you executed in the past as it is usually also not cleared after ending a session.

[1] `https://www.r-project.org`

[2] RStudio is freely available with an open source license from `https://www.rstudio.com/products/rstudio/download/`. The version used in this guide is 1.1.463.

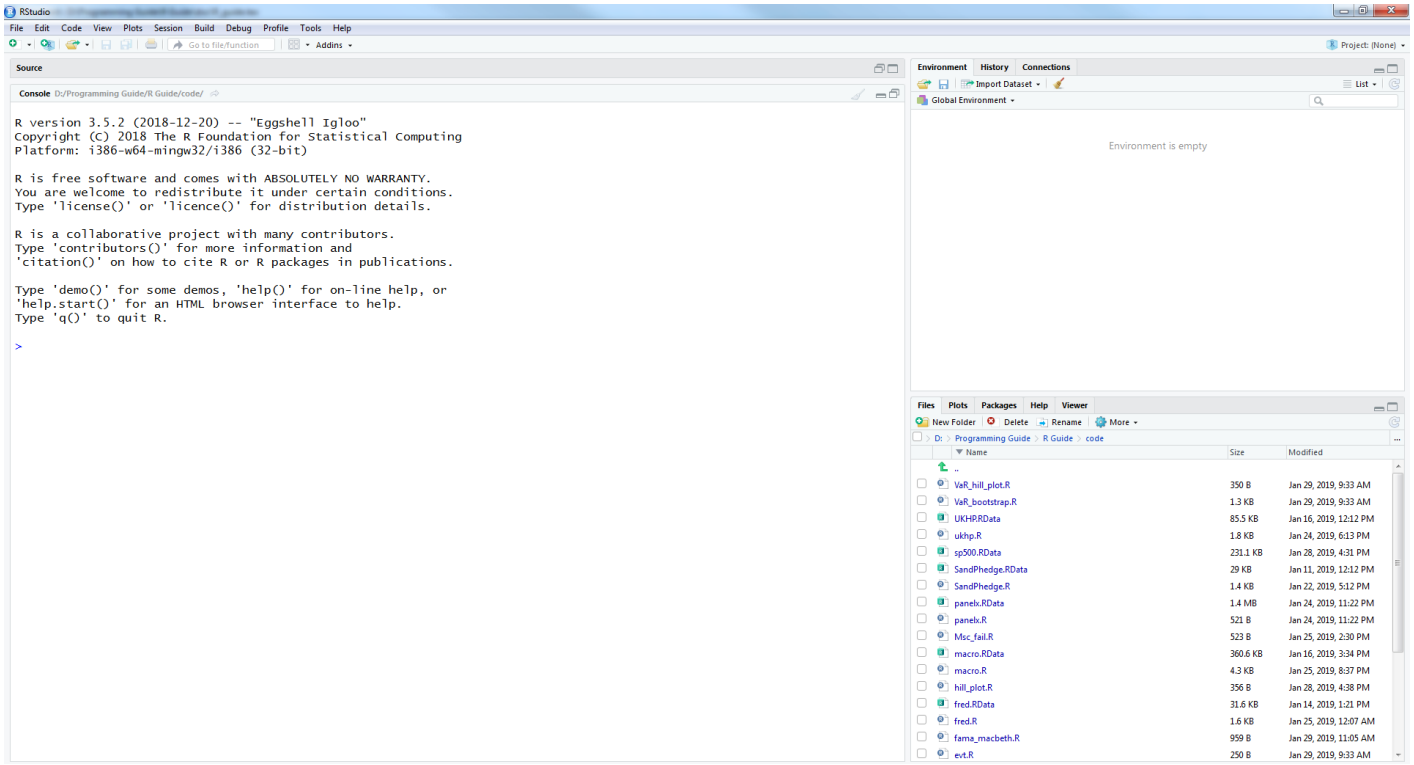


Figure 1: RStudio Main Windows

The *Files* tab shows the directory you are currently working with and the files stored therein. It is helpful to set this directory by default to the folder you want to work in, although this can also be done manually every time you open RStudio. For the general Option, again open **Tools/Global Options...** and change the default working directory in the **General** tab. Alternatively, you can set this working directory every time you restart RStudio by navigating to the directory in the *Files* tab. Once you have open the desired directory, click **More** in the top toolbar of the *Files* tab and select **Set As Working Directory**; after that, you can see an executed command in the *Console* that should resemble the one below with the specific path you chose.

```
> setwd("D:/Programming Guide/R Guide/code")
>
```

This small example is one of the built-in tools where RStudio translates an operation into code and executes it for you, which can helpful in the beginning as you might be overwhelmed by all the commands that you would need to know to do this without the graphical interface. However, note that typing

```
setwd("D:/Programming Guide/R Guide/code")
```

into the *Console* and hitting **Enter** would have had the same effect.

Next to the *Files* tab, you can find the *Plots* tab showing the graphical output produced during the session. The *Packages* window shows which libraries are installed and loaded into the current session memory. Finally, the *Help* tab offers many ways of support and documentation.

2 Getting Started

2.1 Packages

Since R is an open source language, libraries are an essential part of R programming. These are packages of R functions, data or compiled code that have been provided by other users. Some standard libraries are pre-installed, while others have to be added manually. Clicking on the *Packages* tab in RStudio displays a list of installed packages. The Comprehensive R Archive Network (CRAN) is a repository of packages that is widely known and from which packages can be downloaded and installed directly through RStudio. For the first exercises you need to download and install the package **'readxl'** which contains functions to read data from Microsoft Excel '.xls' and '.xlsx' files. To select the necessary package, click on **Tools/Install Packages...** in the main toolbar. The window in Figure 2 appears.

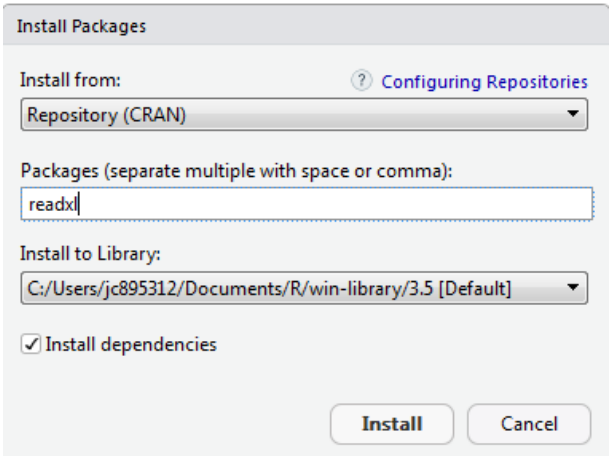


Figure 2: Installing a Package in RStudio

Now specify the package by typing 'readxl'. Note that you can install packages from your hard drive instead of the CRAN repository, too. Make sure the option **Install dependencies** is ticked, so all necessary packages which 'readxl' is linked to are also installed. A log of the installation process is displayed in the *Console*. A package only needs to be installed once, but it is recommended to update package by clicking the **Update** button right of Install in the *Packages* window regularly to keep track of changes. You will see that installing packages is also possible within the *Console* directly using the function **install.packages()**.

However, installing the package does not automatically provide us with the functions included in the package. To include these in a program, they need to be loaded by either using the command **library()**, or, to embed the functions (of, for example, 'readxl') into a program, tick mark 'readxl' in the list of libraries in the *Packages* window. Throughout this guide, we will introduce packages for the different sections that contain functions of use for achieving particular tasks. However, note that many libraries are already part of the System library and do not have to be installed.

2.2 Importing Data

As a first example, we will import the file **'UKHP.xls'** into RStudio. For this purpose, RStudio has a built-in function that translates the action into code. In this way you can learn how to write this code yourself. Click **File/Import Dataset/From Excel...** to open the window in Figure 3. If you have not installed 'readxl' beforehand, RStudio will now ask you to do so as this package is needed to import the dataset.

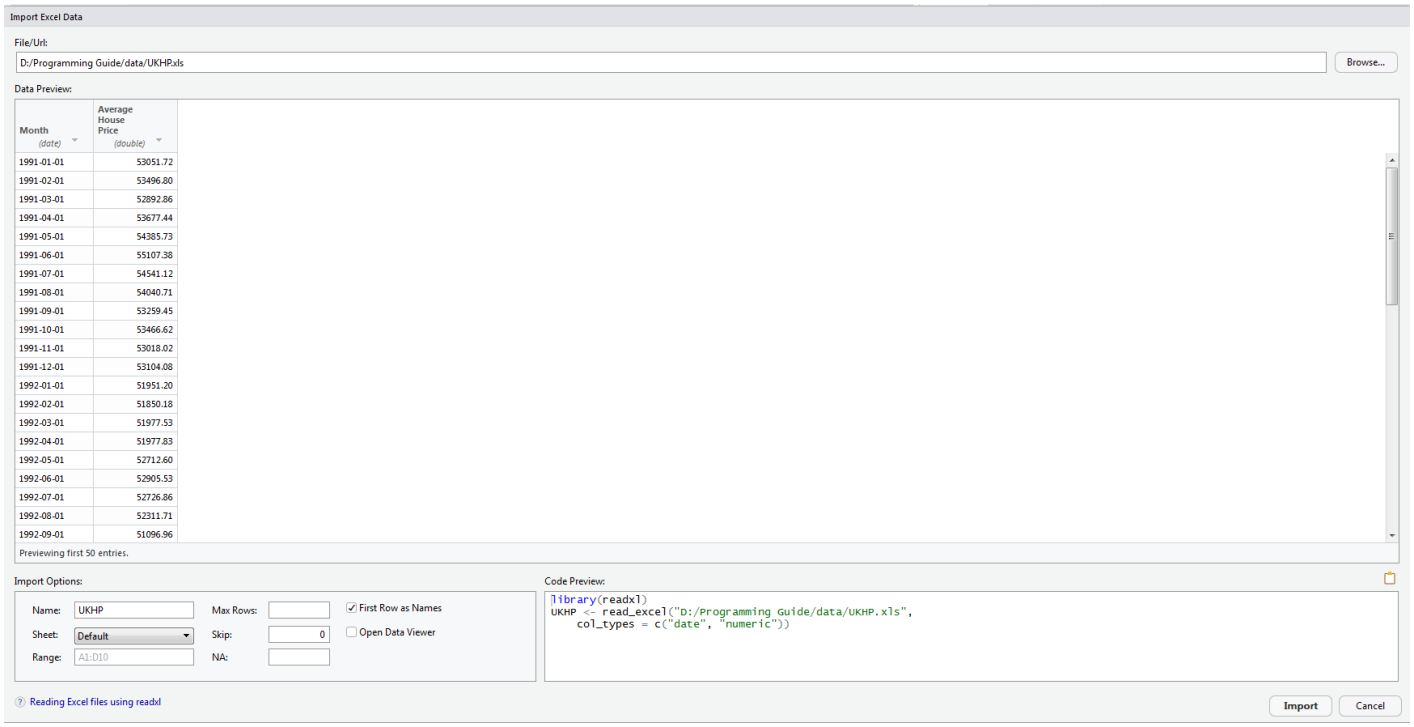


Figure 3: Importing Data

In the Import window, choose the file **UKHP.xls** and you will see a preview of how the file will be imported. The dataset includes two variables, **Month** and **Average House Price**, which are currently both interpreted as doubles. To change the data type of 'Month' to Date, click on the small arrow next to it and choose **Date** in the drop down menu.

Using this embedded tool to import a dataset might not be the most efficient way, but it offers a first glance at how these actions are translated into R code.[3] In the **Code Preview** panel in the lower right corner, the following code is presented to the reader.

```
library(readxl)
UKHP <- read_excel("D:/Programming Guide/data/UKHP.xls",
    col_types = c("date", "numeric"))
View(UKHP)
```

Clicking **Import** will run this code, so it will be displayed in the *Console*. The first line is necessary to embed the package 'readxl' which was installed before and includes the function 'read_excel'. This function is called in the second line to read from the given path. Note that the path is dependent on where the files are stored. To change the format of the first column into date, the argument 'col_types' has to be set to 'date'. However, since there are two columns, both formats have to be specified. By setting 'col_types' to **c("date", "numeric")**. Here another function is used, **c()**, which is a generic function that combines the arguments into a vector. Finally, the data read is stored in the list object **UKHP**. To view the data, type **View(UKHP)** or click on the new element in the *Environment* window.

Within this example, we can already see how much of the R code we will apply in this guide is structured. Most operations are done using functions like **read_excel**. These functions need input arguments, such as a path, and can take optional arguments such as **col_types**.

[3]However, there are several other ways to import data into R depending on the format of the source file, including **read.table()** for '.txt' files, **read.csv()** for '.csv' files which are included in the system library **utils** or even the alternative package **'xlsx'** and its function **read.xlsx** for Excel files.

This example also reveals a peculiarity of R, which originally uses the arrow symbol '<–' to assign values to variables. However, it is also possible to use the more common equal sign '='. Although it might be unusual to use an arrow, it can improve the readability of code, since setting arguments in R is already done by using '=', see 'col_types = ...'. Throughout this guide, the equals sign will be used.

2.3 Data Description

After having loaded the dataset successfully into the workspace, it should appear in the *Environment* tab under **Data** as in Figure 4. Clicking on the blue arrow to the left will unfold the variables.

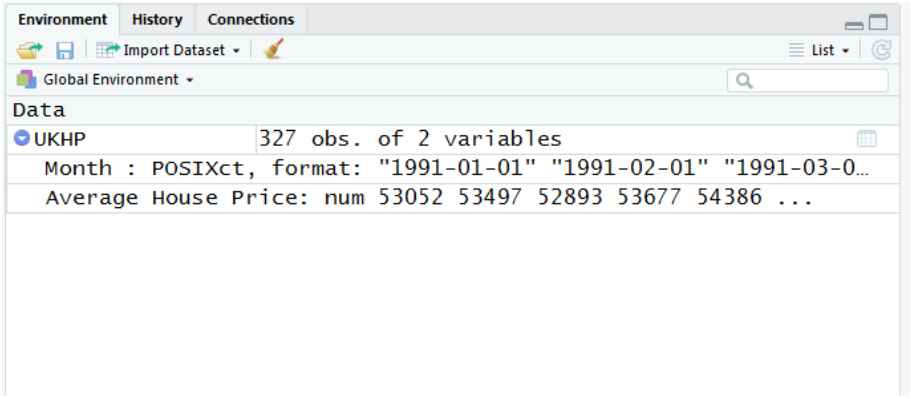


Figure 4: The *Environment* Tab with Imported Data

We can see that the dataset has two variables with 327 observations. The variable **Month** is of format **POSIXct**, which is a specific date and time format, and the variable **Average House Price** is of format **numeric (num)**. If we click on **UKHP** in the *Environment* tab, a new tab will open in the source window showing the dataset. This way ,we can check whether the data have been imported correctly.

Another good idea is to check some basic summary statistics to make sure data has the expected properties. R provides the function **summary()** for this. As there is no built-in function, we have to type this command directly into the *Console*. Typing **summary(UKHP)** into the *Console* and pressing **Enter** will produce the following output

```
> summary(UKHP)
      Month                  Average House Price
 Min.   :1991-01-01 00:00:00   Min.   : 49602
 1st Qu.:1997-10-16 12:00:00   1st Qu.: 61654
 Median :2004-08-01 00:00:00   Median :150946
 Mean   :2004-07-31 17:54:29   Mean   :124660
 3rd Qu.:2011-05-16 12:00:00   3rd Qu.:169239
 Max.   :2018-03-01 00:00:00   Max.   :211756
>
```

While the summary command displays several statistics, they can also be called individually using the specific functions like **mean()**, **median()**, **quantile()**, **min()** or **max()**. There is also **sd()** as a function for the standard deviation. Throughout this guide, these functions will be used frequently and explained in more detail. However, they can be used in a straight-forward fashion in most cases.

Summary statistics for the variable **Month** are not necessary, so to access the column 'Average House Prices' only in R, the extract operator **$** followed by the name of the variable can be used. Hence to

obtain the mean of house price, type **mean(UKHP$'Average House Price')**. The function **names()** provides the names of the variables of a list such as 'UKHP'.[4]

2.4 Changing and Creating Data

In any programming language including R, it is highly recommended not to use white spaces for variable names. While other platforms do not allow for it, when reading data R does permit white spaces to be part of the variable names. However, to avoid future problems, this should be changed. As we learned above, with **names()** the variable names can be accessed. Since there are two variables, **names(UKHP)** will return a vector with two entries. To address specific entries of a vector or matrix in R, you use square bracket '[]'. Hence, we address the second entry of the name vector followed by an equals sign and the new string for the name. We type

```
names(UKHP)[2] = "hp"
```

This command will set the new name of the second column to **hp**. As seen before, strings in R have to be put between double quotes. However, single quotes also work.

Writing a new variable into the data 'UKHP' is fairly simple. Choose the name and define the variable directly afterwards in the same line. For the computation of returns, however, we need some more functions. In the end, the new return series 'dhp' will be defined by typing

```
UKHP$dhp = c(NA, 100*diff(UKHP$hp)/UKHP$hp[1:nrow(UKHP)-1])
```

into the *Console*. But let us go through the right-hand side of this line step by step.

First, we use the function **c()**, which we have seen before, to create vector out of a collection of single values. This is necessary, because when creating returns, one observation is lost automatically, which has to be set to ''NA'. Without this adjustment, R, would produce an error since the new variable does not have the appropriate length to fit into the data frame. For demonstration, try to type the same command without adding an 'NA' before. You should obtain the following output.

```
> UKHP$dhp = 100*diff(UKHP$hp)/UKHP$hp[1:nrow(UKHP)-1]
Error in `$<-.data.frame`(`*tmp*`, dhp, value = c(0.838950425562273, -1.12892201927641,  :
  replacement has 326 rows, data has 327
>
```

Therefore, typing only the definition of returns would not be compatible with the necessary length of the vector. The actual computation of returns is done in

```
100*diff(UKHP$hp)/UKHP$hp[1:nrow(data)-1]
```

which uses two new functions, **diff()** and **nrow()**. The term **diff(UKHP$hp)** computes the vector of differences $(hp_2 - hp_1, hp_3 - hp_2, \ldots,)$, which forms the numerator of the return series. The denominator is formed as **UKHP$hp[1:nrow(data)-1]**, which takes into account that the last entry of 'hp' is not used. To use all values up to the second to last entry of 'hp', the range is set as **1:nrow(UKHP)**, where

[4] If the default settings are not altered, RStudio also provides the possible completions automatically. In this case, after $ one can choose between **Month** and **'Average House Price'**.

nrow() returns the number of rows of 'UKHP'. The colon '**:**' in R can be used to create sequences with step size 1, e.g., the numbers from 1 to 10 are abbreviated by **1:10**.[5]

Note that R automatically understood that the two vectors had to be divided element-wise. This feature is called vectorisation and is very helpful as we do not have to tell R what to do with every single entry. Instead, the syntax is automatically interpreted as an element-wise division. However, this also makes it necessary to think carefully about the dimensions of the variables because R will interpret commands in specific pre-determined ways instead of producing an error. Table 1 gives some more examples for operators that can be used to create or compare variables.

Table 1: Commonly Used Operators

Z/2	Dividing	==	(exactly) equal to
Z*2	Multiplication	!=	not equal to
Z^2 or Z**2	Squaring	>	larger than
log(Z)	Taking the logarithms	<	smaller than
exp(Z)	Taking the exponential	>=	larger than or equal to
sqrt(Z)	Square root	<=	smaller than or equal to
abs(Z)	Absolute value	&	AND
		|	OR
		!	NOT

2.5 Graphics and Plots

This subsection will introduce some of the standard ways to produce graphical output in R using only functions from the system library **graphics**.[6]

While you can customise any plot individually, it is helpful to set certain parameters globally so that they do not have to be adjusted in every single plot. This includes fonts, font sizes, colours, labels, symbols, line styles, legends and many more. To see what is the actual global setting for each of these parameters, type **par()** into the *Console*. You can also view the settings in a list by typing **View(par())**. For explanations of the specific parameters, type **help(par)**.

Let us plot a simple line graph of the house price series. Before we give the instructions for the actual plot produced by the function **plot**, we set the parameters to improve the readability. The following code produces the graph in Figure 5.

```
par(cex.axis = 1.5, cex.lab = 1.5, lwd = 2)
plot(UKHP$Month,UKHP$hp,type = 'l',xlab="Date",ylab="House price")
```

In the first line, we double the size of the axis (**cex.axis**), labels (**cex.lab**) and linewidth (**lwd**) within the **par** command. The second line produces a line graph that shows the date variable 'Month' on the x-axis and the house price 'hp' on the y-axis. Setting the argument **type** to **'l'** will connect the points.[7] With the arguments **xlab** and **ylab** you can write new strings to the axis labels, otherwise the names of the data series are used.

[5]Note that there is also the function **seq()** to construct more complicated sequences. See the *Help* tab for more information, or type **help(seq)** into the *Console*.

[6]One of the most used extensions to the standard graphics is the package **ggplot2**. See the documentation on the CRAN website for more information (`https://cran.r-project.org/web/packages/ggplot2/ggplot2.pdf`).

[7]The default setting here is 'p', which will only plot the points. Search for 'plot' in the *Help* window to receive more information on graphical options for plots.

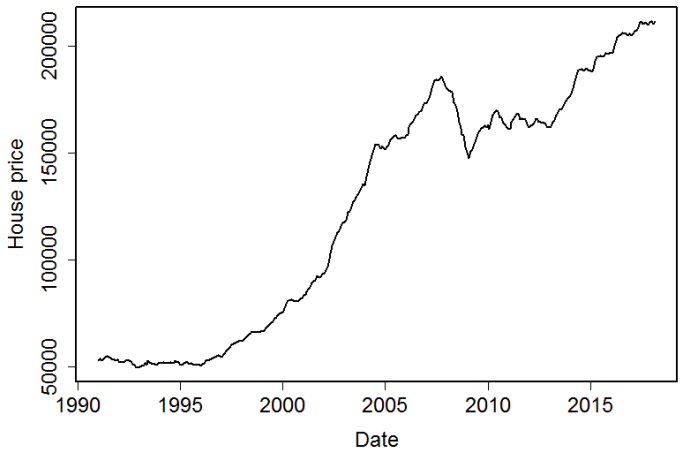


Figure 5: Line Plot of the Average House Price Series

After executing the commands, you will see that the lower right window automatically switches to the *Plots* tab displaying the most recent plot. The toolbar above enables you to export your plots and saves them so that you can navigate through them using the arrows in the tool bar.

To plot histograms, the **graphics** package provides the function **hist**. To produce a histogram of the return series **dhp**, simply type **hist(UKHP$dhp)** and you will find the graph in Figure 6 in the *Plots* tab. If you want to refine the histogram, use the argument **breaks** within the **hist** function.[8]

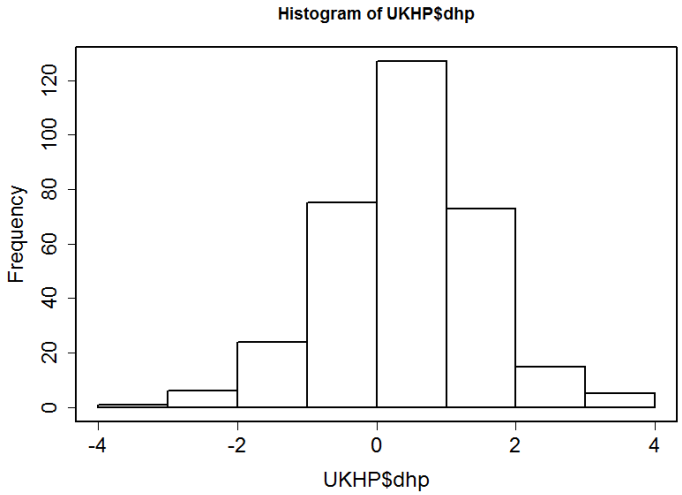


Figure 6: Histogram of Housing Return Series

2.6 Keeping Track of Your Work

In order to be able to reproduce your work and remember what you have done after some time, you can save the current workspace by clicking **Session/Save Workspace As...** in the top menu. Save the workspace as **UKHP.RData**. The instructions are logged in the *History* window, the second tab in the upper right corner. However, it is easier to save the instruction in an R script or '.R'-file. To create a new R script, click **File/New File/R Script**. In the example below you will find all of the instructions that we have used so far.

[8]Note that I also added a box around the graph with the simple command **box()**.

```
library(readxl)
UKHP = read_excel("D:/Programming Guide/data/UKHP.xls", col_types = c("date"
    , "numeric"))
names(UKHP)[2] = "hp"
UKHP$dhp = c(NA, 100*diff(UKHP$hp)/UKHP$hp[1:nrow(UKHP)-1])
par(cex.axis = 2, cex.lab =2, lwd = 2)
plot(UKHP$Month,UKHP$hp,type = 'l',xlab="Date",ylab="House price")
hist(UKHP$dhp)
```

This file can again be saved clicking **File/Save**. Save the script as **UKHP.R**. From an empty workspace, you could regain all of the results by running the script. To do so, open the file with **File/Open File...**, and in the script run any line by clicking **Run** in the top right corner of the *Source* window or by pressing **Ctrl+Enter** with the cursor is the respective line. To run the whole block of code at once, either mark everything or click **Source** in the upper right-hand corner of the *Source* window.

However, since we already saved the results in the file '**UKHP.RData**', we don't necessarily need to re-run the code. You can import the results by clicking **File/Open File...** and choosing the '.RData' file. As you see in the *Console*, this command can be typed using the function **load**. To avoid naming the whole path, it is useful to set the working directory to the directory where you keep your files, then a simple **load("".RData"** is sufficient.[9]

[9]See the beginning of this chapter to check how to set your working directory.

3 Linear Regression – Estimation of an Optimal Hedge Ratio

Reading: Brooks (2019, Section 3.3)

This section shows how to run a bivariate regression in R. In our example, an investor wishes to hedge a long position in the S&P500 (or its constituent stocks) using a short position in futures contracts. The investor is interested in the optimal hedge ratio, i.e., the number of units of the futures asset to sell per unit of the spot assets held.[10]

This regression will be run using the file 'SandPhedge.xls', which contains monthly returns for the S&P500 Index (in Column 2) and the S&P500 Futures (in Column 3). Before we run the regression, this Excel workfile needs to be imported. As learned in the section before, this can be done using **File/Import Dataset/From Excel** or directly by typing the line of code

```
SandPhedge <- read_excel("D:/Programming Guide/data/SandPhedge.xls",
   col_types = c("date", "numeric", "numeric"))
```

Note that the path of the file has to be adapted to the local path of course. Also, the format of the first variable has been changed to date. Do not forget to tick-mark the package 'readxl' or insert the line **library(readxl)** before importing the data.

We can see the three imported variables in the dataset 'SandPhedge' in the *Environment* window on the right-hand side of the screen. In order to examine the data and verify some data entries, you can click on 'SandPhedge' or type **View(SandPhedge)** into the *Console*.

We run our analysis based on returns to the S&P500 index instead of price levels; so the next step is to transform the 'spot' and 'future' price series into percentage returns. For our analysis, we use continuously compounded returns, i.e., logarithmic returns, instead of simple returns as is common in academic finance research. To create the log return series for spot and futures prices, type

```
SandPhedge$rspot = c(NA,100*diff(log(SandPhedge$Spot)))
SandPhedge$rfutures = c(NA,100*diff(log(SandPhedge$Futures)))
```

To have a look at the new variables, use the **summary** command. Since you do not need to see a summary of every variable in the dataset, specify the variables that should be summarised. Type

```
summary(SandPhedge[c("rspot","rfutures")])
```

and the following output should be displayed in the *Console*.

```
> summary(SandPhedge[c("rspot","rfutures")])
     rspot              rfutures
 Min.   :-18.5636   Min.   :-18.9447
 1st Qu.: -1.8314   1st Qu.: -1.9314
 Median :  0.9185   Median :  0.9976
 Mean   :  0.4168   Mean   :  0.4140
 3rd Qu.:  3.2765   3rd Qu.:  3.1336
 Max.   : 10.2307   Max.   : 10.3872
 NA's   :1          NA's   :1
>
```

[10] See also Chapter 9 in Brooks (2019).

We observe that the two return series are quite similar as based on their mean values, standard deviations (not displayed here), as well as minimum and maximum values, as one would expect from economic theory.

Now proceed to estimate the regression. Note that it is not necessary to tell R which of the three variables is the time indicator. We can run a regression directly after having created the new variables. To estimate linear models, R provides the function **lm()**. The main argument **lm()** requires is the formula for the regression given by $y \sim x$, with y as dependent and x as independent variable. The independent variable is the **rfutures** series. Thus, we are trying to explain variations in the spot rate with variations in the futures rate. An intercept is automatically included. We specify the regression model and save it as **lm_returns** in the first line and then call the summary function to present the results by typing

```
lm_returns = lm(rspot ~ rfutures,data = SandPhedge)
summary(lm_returns)
```

Note that instead of entering the variables directly as **SandPhedge$rspot** and **SandPhedge$rspot**, **lm()** provides the argument data such that we only need to type the variable names.[11] After running these two lines of code, the output below will be presented in the console.

```
> lm_returns = lm(rspot ~ rfutures,data = SandPhedge)
> summary(lm_returns)

Call:
lm(formula = rspot ~ rfutures, data = SandPhedge)

Residuals:
     Min       1Q   Median       3Q      Max
-2.45284 -0.16401  0.00236  0.23692  2.33789

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 0.013077   0.029473   0.444    0.658
rfutures    0.975077   0.006654 146.543   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4602 on 244 degrees of freedom
  (1 observation deleted due to missingness)
Multiple R-squared:  0.9888,    Adjusted R-squared:  0.9887
F-statistic: 2.147e+04 on 1 and 244 DF,  p-value: < 2.2e-16

>
```

The parameter estimates for the intercept ($\hat{\alpha}$) and slope ($\hat{\beta}$) are 0.013 and 0.975, respectively. A large number of other statistics are also presented in the regression output – the purpose and interpretation of these will be discussed later.

Now we estimate a regression for the levels of the series rather than the returns (i.e., we run a regression of 'Spot' on a constant and 'Futures') and examine the parameter estimates. We follow the steps described above and specify 'Spot' as the dependent variable and 'Futures' as the independent variable by typing and running the commands

```
lm_prices = lm(Spot ~ Futures,data = SandPhedge)
summary(lm_prices)
```

[11] A discussion of other optional arguments of **lm()** will follow in later sections.

The intercept estimate ($\hat{\alpha}$) in this regression is –2.838 and the slope estimate ($\hat{\beta}$) is 1.002, as presented in the output below.

```
> lm_prices = lm(Spot ~ Futures,data = SandPhedge)
> summary(lm_prices)

Call:
lm(formula = Spot ~ Futures, data = SandPhedge)

Residuals:
    Min      1Q  Median      3Q     Max
-64.576  -1.996   1.436   4.309  16.612

Coefficients:
             Estimate Std. Error  t value Pr(>|t|)
(Intercept) -2.8378335  1.4889725   -1.906   0.0578 .
Futures      1.0016065  0.0009993 1002.331   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 6.908 on 245 degrees of freedom
Multiple R-squared:  0.9998,    Adjusted R-squared:  0.9998
F-statistic: 1.005e+06 on 1 and 245 DF,  p-value: < 2.2e-16

>
```

Let us now turn to the (economic) interpretation of the parameter estimates from both regressions. The estimated return regression slope parameter measures the optimal hedge ratio as well as the short-run relationship between the two series. By contrast, the slope parameter in a regression using the raw spot and futures indices (or the log of the spot series and the log of the futures series) can be interpreted as measuring the long-run relationship between them. The intercept of the price level regression can be considered to approximate the cost of carry. Looking at the actual results, we find that the long-term relationship between spot and futures prices is almost 1:1 (as expected). Before exiting RStudio, do not forget to click **Save** to place the workspace into **SandPhedge.RData**.

4 Hypothesis Testing – Example 1: Hedging Revisited

Reading: Brooks (2019, Sections 3.8 and 3.9)

Let us now have a closer look at the results table from the returns regressions in the previous section where we regressed S&P500 spot returns on futures returns in order to estimate the optimal hedge ratio for a long position in the S&P500. If you closed RStudio, reload the workspace **'SandPhedge.RData'** by clicking **File/Open File...**. The workspace should include the dataset 'SandPhedge' consisting of five variables and the two linear models. While we have so far mainly focused on the coefficient estimates, i.e., α and β, the two linear models also contain several other statistics which are presented next to the coefficient estimates: standard errors, the t-ratios and the p-values.

The t-ratios are presented in the third column indicated by 't value' in the column heading. They are the test statistics for a test of the null hypothesis that the true values of the parameter estimates are zero against a two-sided alternative, i.e., they are either larger or smaller than zero. In mathematical terms, we can express this test with respect to our coefficient estimates as testing $H_0 : \alpha = 0$ versus $H_1 : \alpha \neq 0$ for the constant '_cons' in the second row of numbers and $H_0 : \beta = 0$ versus $H_1 : \beta \neq 0$ for 'rfutures' in the first row. Let us focus on the t-ratio for the α estimate first. We see that with a value of only 0.44, the t-ratio is very small, which indicates that the corresponding null hypothesis $H_0 : \alpha = 0$ is likely not to be rejected. Turning to the slope estimate for 'rfutures', the t-ratio is high with 146.54 suggesting that $H_0 : \beta = 0$ is to be rejected against the alternative hypothesis of $H_1 : \beta \neq 0$. The p-values presented in the fourth column, '$P > |t|$', confirm our expectations: the p-value for the constant is considerably larger than 0.1, meaning that the corresponding t-statistic is not even significant at the 10% level; in comparison, the p-value for the slope coefficient is zero to, at least, three decimal places. Thus, the null hypothesis for the slope coefficient is rejected at the 1% level.

While R automatically computes and reports the test statistics for the null hypothesis that the coefficient estimates are zero, we can also test other hypotheses about the values of these coefficient estimates. Suppose that we want to test the null hypothesis that $H_0 : \beta = 1$. We can, of course, calculate the test statistics for this hypothesis test by hand; however, it is easier if we let a function do this work. The package **car** contains the function **linearHypothesis()** which can do this job. Hence we install and load the package from the *Packages* window. Looking into the help function for this package, you can find the documentation for the whole package and the function by clicking on the package in the *Packages* window. As arguments, **linearHypothesis** asks for a model, while the second input is a vector of restrictions that can be written straightforwardly using the variable names. Since we want to specify $\beta = 1$, we set

```
linearHypothesis(lm_returns,c("rfutures=1"))
```

After running the code, the output on the following page should appear in the *Console*. The first lines list the models that are being compared: the restricted model 1, where $\beta = 1$ and the original model 'rspot $\sim$ rfutures'. Below, we find the F-test statistics under 'F'; 14.03 states the value of the F-test. The corresponding p-value is 0.0002246, as stated in the last column. As it is considerably smaller than 0.01, we can reject the null hypothesis that the coefficient estimate is equal to 1 at the 1% level.

```
> linearHypothesis(lm_returns,c("rfutures=1"))
Linear hypothesis test

Hypothesis:
rfutures = 1

Model 1: restricted model
Model 2: rspot ~ rfutures

  Res.Df    RSS Df Sum of Sq     F    Pr(>F)
1    245 54.656
2    244 51.684  1    2.9718 14.03 0.0002246 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
>
```

We can also perform hypothesis testing on the levels regressions. For this we use the regression in levels **lm_prices** and type this command into the *Console*:

```
linearHypothesis(lm_prices,c("Futures=1"))
```

The following output is presented.

```
> linearHypothesis(lm_prices,c("Futures=1"))
Linear hypothesis test

Hypothesis:
Futures = 1

Model 1: restricted model
Model 2: Spot ~ Futures

  Res.Df   RSS Df Sum of Sq      F Pr(>F)
1    246 11816
2    245 11693  1    123.35 2.5846 0.1092
>
```

With an F-statistic of 2.58 and a corresponding p-value of 0.1092, we find that the null hypothesis is not rejected at the 5% significance level.

5 Hypothesis Testing – Example 2: The CAPM

Reading: Brooks (2019, Sections 3.10 and 3.11)

This exercise will estimate and test some hypotheses about the CAPM beta for several US stocks. The data for this example are contained in the excel file 'capm.xls'. We first import this data file by selecting **File/Import Dataet/From Excel...**. Without further specifying anything, R automatically converts the first column into a date variable. The imported data file contains monthly stock prices for the S&P500 index ('SANDP'), the four companies, Ford ('FORD'), General Electric ('GE'), Microsoft ('MICROSOFT') and Oracle ('ORACLE'), as well as the 3-month US-Treasury bill yield series ('USTB3M') from January 2002 until February 2018. Before proceeding to the estimation, save the R workspace as 'capm.RData' by selecting **Session/Save workspace as...** .

It is standard in the academic literature to use five years of monthly data for estimating betas, but we will use all of the observations (over fifteen years) for now. In order to estimate a CAPM equation for the Ford stock for example, we need to first transform the price series into (continuously compounded) returns and then to transform the returns into excess returns over the risk-free rate.

To generate continuously compounded returns for the S&P500 index, type

```
capm$rsandp = c(NA,100*diff(log(capm$SANDP)))
```

and returns are written to the new variable **rsandp** in the list **capm**. Recall that the first entry needs to be empty ('NA') as the returns series has one observation less.

We need to repeat these steps for the stock prices of the four companies. Hence,type the commands

```
capm$rford = c(NA,100*diff(log(capm$FORD)))
capm$rge = c(NA,100*diff(log(capm$GE)))
capm$rmsoft = c(NA,100*diff(log(capm$MICROSOFT)))
capm$roracle = c(NA,100*diff(log(capm$ORACLE)))
```

into the *Console* and press **Enter**. We should then find the new variables in the *Environment* window after clicking on 'capm'.

In order to transform the returns into excess returns, we need to deduct the risk free rate, in our case the 3-month US-Treasury bill rate, from the continuously compounded returns. However, we need to be slightly careful because the stock returns are monthly, whereas the Treasury bill yields are annualised. When estimating the model it is not important whether we use annualised or monthly rates; however, it is crucial that all series in the model are measured consistently, i.e., either all of them are monthly rates or all are annualised figures. We decide to transform the T-bill yields into monthly figures, before generating excess returns:

```
capm$USTB3M = capm$USTB3M/12
capm$ersandp = capm$rsandp - capm$USTB3M
```

We similarly generate excess returns for the four stock returns. Before running the CAPM regression, we plot the data series to examine whether they appear to move together. We do this for the S&P500 and the Ford series. The following two lines will produce the graph in Figure 7

```
par(cex.axis = 1.5, cex.lab = 1.5, lwd = 2)
plot(capm$Date,capm$ersandp,type='l',col = "red",ylim=c(-100,100),ylab="")
lines(capm$Date,capm$erford, lwd = 1)
```

```
legend("topright",c("SP500","Ford"), col = c("red","black"),lty=1,cex=1.5)
```

Since both plots should appear in the same window, some modifications of the simple **plot()** command are necessary. To insert a second plot into an existing one can be achieved with the function **lines()**. However, to make it easier to distinguish between the two lines, we also need to change the colour as otherwise both lines would be black. This is done by setting the argument **col** to "red". Further, we increase the linewidth (**lwd**) to 1 and adjust the limits of the y-axis (**ylim**) to include the range from -100 to 100, so all points of the second plot are visible in the graph. With the **legend** command, we can further customise the legend.

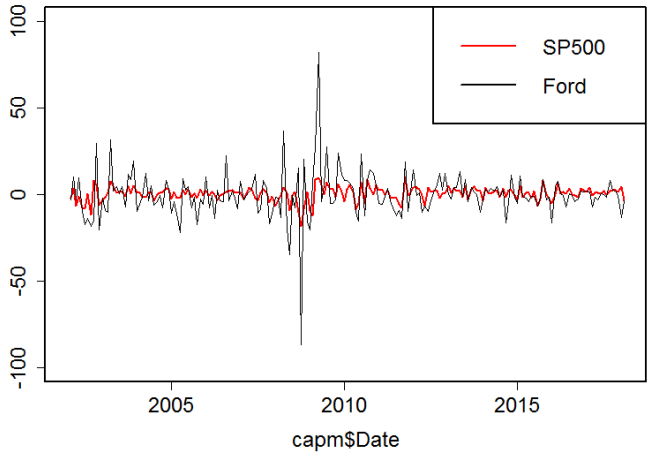


Figure 7: Time-series Plot of Two Series

However, in order to get an idea about the association between two series, a scatter plot might be more informative. To generate a scatter plot, use **plot()** with 'ersandp' as x-axis and 'erford' as y-axis.

```
plot(capm$ersandp,capm$erford,pch=19)
```

To improve visibility, we also set the argument **pch** for the point symbols to a solid circle (19).[12]

[12] A list with all possible symbols can be found in the help documentation for the function **points()**.

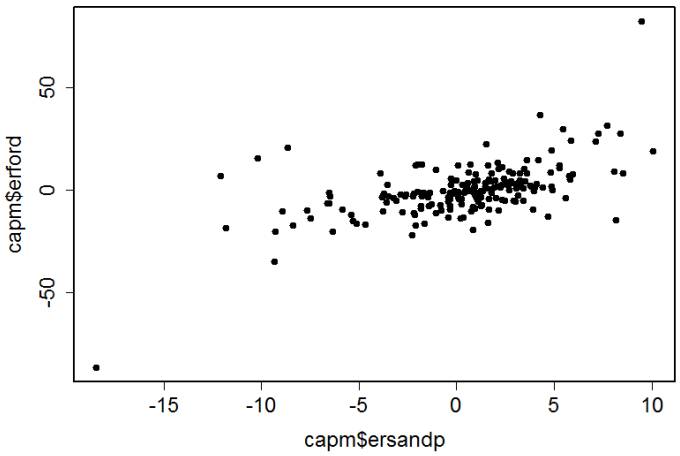


Figure 8: Generating a Scatter Plot of Two Series

We see from this scatter plot that there appears to be a weak association between 'ersandp' and 'erford'. We can also create similar scatter plots for the other data series and the S&P500.

For the case of the Ford stock, the CAPM regression equation takes the form

$$(R_{Ford} - r_f)_t = \alpha + \beta(R_M - r_f)_t + u_t \tag{1}$$

Thus, the dependent variable (y) is the excess return of Ford 'erford' and it is regressed on a constant as well as the excess market return 'ersandp'. Hence, specify the CAPM equation regression as

```
lm_capm = lm(erford ~ ersandp, data = capm)
summary(lm_capm)
```

and run this code to obtain the results below.

```
> lm_capm = lm(erford ~ ersandp, data = capm)
> summary(lm_capm)

Call:
lm(formula = erford ~ ersandp, data = capm)

Residuals:
    Min      1Q  Median      3Q     Max
-50.727  -5.027  -1.080   3.482  65.145

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.9560     0.7931  -1.205     0.23
ersandp       1.8898     0.1916   9.862   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 10.98 on 191 degrees of freedom
  (1 observation deleted due to missingness)
Multiple R-squared:  0.3374,    Adjusted R-squared:  0.3339
F-statistic: 97.26 on 1 and 191 DF,  p-value: < 2.2e-16

>
```

Take a couple of minutes to examine the results of the regression. What is the slope coefficient estimate and what does it signify? Is this coefficient statistically significant? The beta coefficient (the slope coefficient) estimate is 1.89 with a t-ratio of 9.86 and a corresponding p-value of 0.000. This suggests that the excess return on the market proxy has highly significant explanatory power for the variability of the excess return of Ford stock.

Let us turn to the intercept now. What is the interpretation of the intercept estimate? Is it statistically significant? The α estimate is –0.96 with a t-ratio of –1.21 and a p-value of 0.230. Thus, we cannot reject the null hypothesis that the α estimate is different from zero, indicating that the Ford stock does not seem to significantly outperform or underperform the overall market.

Assume that we want to test the null hypothesis that the value of the population coefficient on 'ersandp' is equal to 1. How can we achieve this? The answer is to run an F-test using **linearHypothesis()** from the **car** package. Load the package by typing **library(car)** and then run the code

```
linearHypothesis(lm_capm,c("ersandp=1"))
```

The output should appear as below.

```
> linearHypothesis(lm_capm,c("ersandp=1"))
Linear hypothesis test

Hypothesis:
ersandp = 1

Model 1: restricted model
Model 2: erford ~ ersandp

  Res.Df   RSS Df Sum of Sq     F    Pr(>F)
1    192 25618
2    191 23020  1    2598.5 21.56 6.365e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
>
```

The F-statistic of 21.56 with a corresponding p-value of 0.000006 implies that the null hypothesis of the CAPM beta of the Ford stock being 1 is convincingly rejected and hence the estimated beta of 1.89 is significantly different from 1.[13]

[13]This is hardly surprising given the distance between 1 and 1.89. However, it is sometimes the case, especially if the sample size is quite small and this leads to large standard errors, that many different hypotheses will all result in non-rejection – for example, both $H_0 : \beta = 0$ and $H_0 : \beta = 1$ not rejected.

6 Sample Output for Multiple Hypothesis Tests

Reading: Brooks (2019, Section 4.4)

This example uses the **'capm.RData'** dataset constructed in the previous section. So in case you are starting a new session, re-load the workspace. If we examine the F-statistic of 97.26 from the regression **lm_capm**), this also shows that the regression slope coefficient is significantly different from zero, which in this case is exactly the same result as the t-test (t-stat: 9.86) for the beta coefficient. Since in this instance there is only one slope coefficient, the F-test statistic is equal to the square of the slope t-ratio.

Now suppose that we wish to conduct a joint test that both the intercept and slope parameters are one. We would perform this test in a similar way to the test involving only one coefficient. We can specify the restrictions in matrix form by saying that the product of coefficients with the identity matrix should equal the vector $(1, 1)'$. Therefore, we use the arguments **hypothesis.matrix** and **rhs** of **linearHypothesis**. We type in the *Console*:

```
linearHypothesis(lm_capm,hypothesis.matrix = diag(2),rhs = c(1,1))
```

This command shows the generic function **diag()**, which can be used to create diagonal matrices. Applied on a scalar, it returns the identity matrix of that dimension, but it can also be used with a vector that specifies the diagonal of the matrix – i.e., **diag(2)** and **diag(c(1,1))** are equivalent commands.

```
> linearHypothesis(lm_capm,hypothesis.matrix = diag(2),rhs = c(1,1))
Linear hypothesis test

Hypothesis:
(Intercept) = 1
ersandp = 1

Model 1: restricted model
Model 2: erford ~ ersandp

  Res.Df   RSS Df Sum of Sq      F    Pr(>F)
1    193 26140
2    191 23020  2      3120 12.944 5.349e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
>
```

In the *Output* window above, R produces the familiar output for the F-test. However, we note that the joint hypothesis test is indicated by the two conditions that are stated, '(Intercept) = 1' and 'ersandp = 1'.[14] Looking at the value of the F-statistic of 12.94 with a corresponding p-value of 0.0000, we conclude that the null hypothesis, $H_0 : \beta_1 = 1$ and $\beta_2 = 1$, is strongly rejected at the 1% significance level.

[14] Thus, the command **linearHypothesis(lm_capm,c("(Intercept)=1","ersandp=1"))** is equivalent to the version using two arguments.

7 Multiple Regression Using an APT-Style Model

Reading: Brooks (2019, Section 4.4)

The following example will show how we can extend the linear regression model introduced in the previous sections to estimate multiple regressions in R. In the spirit of arbitrage pricing theory (APT), we will examine regressions that seek to determine whether the monthly returns on Microsoft stock can be explained by reference to unexpected changes in a set of macroeconomic and financial variables. For this, we rely on the dataset **'macro.xls'** which contains nine data series of financial and economic variables as well as a date variable spanning the time period from March 1986 until March 2018 (i.e., 385 monthly observations for each of the series). In particular, the set of financial and economic variables comprises the Microsoft stock price, the S&P500 index value, the consumer price index, an industrial production index, Treasury bill yields for three months and ten years, a measure of 'narrow' money supply, a consumer credit series, and a 'credit spread' series. The latter is defined as the difference in annualised average yields between a portfolio of bonds rated AAA and a portfolio of bonds rated BAA.

Before we can start with our analysis, we need to import the dataset **'macro.xls'** into a new workspace that we save as **'macro.RData'**. The first stage is to generate a set of changes or differences for each of the variables, since the APT posits that the stock returns can be explained by reference to the unexpected changes in the macroeconomic variables rather than their levels. The unexpected value of a variable can be defined as the difference between the actual (realised) value of the variable and its expected value. The question then arises about how we believe that investors might have formed their expectations, and while there are many ways to construct measures of expectations, the easiest is to assume that investors have naive expectations that the next period value of the variable is equal to the current value. This being the case, the entire change in the variable from one period to the next is the unexpected change (because investors are assumed to expect no change).[15]

To transform the variables, directly type the commands into the *Console*:

```
macro$dspread = c(NA,diff(macro$BMINUSA))
macro$dcredit = c(NA,diff(macro$CCREDIT))
macro$dprod = c(NA,diff(macro$INDPRO))
macro$dmoney = c(NA,diff(macro$M1SUPPLY))
macro$inflation = c(NA,diff(log(macro$CPI)))
macro$rterm = c(NA,diff(macro$USTB10Y-macro$USTB3M))
macro$dinflation = c(NA,100*diff(macro$inflation))

macro$rsandp = c(NA,100*diff(log(macro$SANDP)))
macro$ermsoft = c(NA,100*diff(log(macro$MICROSOFT)))-macro$USTB3M/12
macro$ersandp = macro$rsandp-macro$USTB3M/12
```

The final three of these commands calculate returns on the index and the excess returns for the stock and for the index. We can now run the multiple regression using the same function as before, **lm()**. The variable whose behaviour we seek to explain is the excess return of the Microsoft stock, hence put 'ermsoft' on the left-hand side. The explanatory variables are the excess market return (**ersandp**) as well as unexpected changes in: industrial production (**dprod**), consumer credit (**dcredit**), the inflation rate (**dinflation**), the money supply (**dmoney**), the credit spread (**dspread**), and the term spread

[15] It is an interesting question as to whether the differences should be taken on the levels of the variables or their logarithms. If the former, we have absolute changes in the variables, whereas the latter would lead to proportionate changes. The choice between the two is essentially an empirical one, and this example assumes that the former is chosen, apart from for the stock price series themselves and the consumer price series.

(**rterm**). These variables are put on the right-hand side connected with a plus sign.

```
lm_msoft = lm(ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney +
   dspread + rterm, data = macro)
```

Again, run the summary function afterwards to obtain the output below.

```
> lm_msoft=lm(ermsoft~ersandp + dprod + dcredit + dinflation + dmoney + dspread + rterm,data=macro)
> summary(lm_msoft)

Call:
lm(formula = ermsoft ~ ersandp + dprod + dcredit + dinflation +
    dmoney + dspread + rterm, data = macro)

Residuals:
    Min      1Q  Median      3Q     Max
-36.075  -4.440  -0.403   4.616  24.480

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.326002   0.475481   2.789  0.00556 **
ersandp      1.280799   0.094354  13.574  < 2e-16 ***
dprod       -0.303032   0.736881  -0.411  0.68113
dcredit     -0.025364   0.027149  -0.934  0.35078
dinflation   2.194670   1.264299   1.736  0.08341 .
dmoney      -0.006871   0.015568  -0.441  0.65919
dspread      2.260064   4.140284   0.546  0.58548
rterm        4.733069   1.715814   2.758  0.00609 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.845 on 375 degrees of freedom
  (2 observations deleted due to missingness)
Multiple R-squared:  0.3452,    Adjusted R-squared:  0.333
F-statistic: 28.24 on 7 and 375 DF,  p-value: < 2.2e-16

>
```

Take a few minutes to examine the main regression results. Which of the variables has a statistically significant impact on the Microsoft excess returns? Using your knowledge of the effects of the financial and macroeconomic environment on stock returns, examine whether the coefficients have their expected signs and whether the sizes of the parameters are plausible. The regression F-statistic (last line) takes a value of 28.24. Remember that this tests the null hypothesis that all of the slope parameters are jointly zero. The p-value of <2.2e-16 afterwards shows that this null hypothesis should be rejected.

However, there are a number of parameter estimates that are not significantly different from zero – specifically those on the 'dprod', 'dcredit', 'dmoney' and 'dspread' variables. Let us test the null hypothesis that the parameters on these four variables are jointly zero using an F-test. Again we use the function **linearHypothesis**, this time specifying the restrictions using the names of the variables.[16]

```
library(car)
linearHypothesis(lm_msoft,c("dprod=0","dcredit=0","dmoney=0","dspread=0"))
```

Running the above code results in the following output.

[16]Note that the same results can be produced using the arguments **hypothesis.matrix** and **rhs**. However, you need to be careful, as the matrix has to only address these entries. Therefore, the above version is much easier to read.

```
> linearHypothesis(lm_msoft,c("dprod=0","dcredit=0","dmoney=0","dspread=0"))
Linear hypothesis test

Hypothesis:
dprod = 0
dcredit = 0
dmoney = 0
dspread = 0

Model 1: restricted model
Model 2: ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney + dspread +
    rterm

  Res.Df   RSS Df Sum of Sq      F Pr(>F)
1    379 23180
2    375 23078  4    101.88 0.4139 0.7986
>
```

The resulting F-test statistic follows an $F(4, 375)$ distribution as there are 4 restrictions, 383 usable observations and eight parameters to estimate in the unrestricted regression. The F-statistic value is 0.4139 with p-value 0.7986, suggesting that the null hypothesis cannot be rejected. The parameters on 'rterm' and 'dinflation' are significant at the 10% level. Hence they are not included in this F-test and the variables are retained.

7.1 Stepwise Regression

There are a number of different stepwise regression procedures, but the simplest is the uni-directional forwards method. This starts with no variables in the regression (or only those variables that are always required by the researcher to be in the regression) and then it compares the same model with all possible models that include one more variable with respect to the Akaike Information Criterion (AIC). The variable that improves the criterion the most is added to the model and this becomes the new base line model from which the next variable can be added until there are no more variables left or the existing model is superior to all extensions.

We want to conduct a stepwise regression which will automatically select the most important variables for explaining the variations in Microsoft stock returns. As the maximal model, we use the already estimated model **lm_msoft**, which includes an intercept and seven variables. The model to start with will only include an intercept. We therefore estimate **lm_start** as

```
lm_start = lm(ermsoft~1,data=macro[-2,])
step(lm_start,direction = "forward",scope = formula(lm_msoft))
```

Note that we do not use the first two observations since the maximal model also discards these observations as not all variables are defined. For the stepwise regression procedure, we use the function **step()** from the package **stats** which is included in the system library and therefore does not need to be installed.[17] The arguments for **step** include the minimal model which was just estimated only including an intercept, the argument **direction** which we set to "forward", since the model is to be extended, and **scope** which defines the maximal model, which is given using the function **formula**, so it is not necessary to specify it separately. Executing the code yields the following output.

[17]There is also an implementation of the stepwise regression procedure in the **leaps** package.

```
> lm_start = lm(ermsoft~1,data=macro[-2,])
> step(lm_start,direction = "forward",scope = formula(lm_msoft))
Start:  AIC=1733.94
ermsoft ~ 1

             Df Sum of Sq   RSS    AIC
+ ersandp     1   11367.2 23878 1586.8
+ dspread     1     366.8 34878 1731.9
+ rterm       1     303.0 34942 1732.6
+ dinflation  1     219.9 35025 1733.5
<none>                    35245 1733.9
+ dprod       1      78.8 35166 1735.1
+ dcredit     1      10.8 35234 1735.8
+ dmoney      1       1.0 35244 1735.9

Step:  AIC=1586.81
ermsoft ~ ersandp

             Df Sum of Sq   RSS    AIC
+ rterm       1    497.12 23381 1580.8
+ dinflation  1    227.20 23650 1585.2
<none>                    23878 1586.8
+ dcredit     1     70.09 23808 1587.7
+ dprod       1     50.24 23827 1588.0
+ dmoney      1     24.10 23854 1588.4
+ dspread     1      0.13 23878 1588.8

Step:  AIC=1580.75
ermsoft ~ ersandp + rterm

             Df Sum of Sq   RSS    AIC
+ dinflation  1   200.479 23180 1579.5
<none>                    23381 1580.8
+ dcredit     1    69.399 23311 1581.6
+ dprod       1    37.212 23343 1582.1
+ dmoney      1    25.316 23355 1582.3
+ dspread     1     0.048 23380 1582.8

Step:  AIC=1579.45
ermsoft ~ ersandp + rterm + dinflation

          Df Sum of Sq   RSS    AIC
<none>                  23180 1579.5
+ dcredit  1    63.785 23116 1580.4
+ dprod    1    17.067 23163 1581.2
+ dmoney   1    13.831 23166 1581.2
+ dspread  1    12.788 23167 1581.2

Call:
lm(formula = ermsoft ~ ersandp + rterm + dinflation, data = macro[-2,
    ])

Coefficients:
(Intercept)      ersandp        rterm   dinflation
      1.021        1.266        4.739        2.187

>
```

The output is structured into four blocks which represent the four steps until no further variable is added. A block always starts with the AIC value of the prevalent baseline model noted below, hence in the first block is the minimal model, **ermsoft** $\sim$ **1**, which only includes an intercept. The table below specifies all possible extensions of one variable sorted with increasing AIC. Note that there is also the possibility of not adding any variable, denoted by <**none**>. In the first table, clearly adding **ersandp**

to the model improves the AIC the most, it declines from 1733.94 for the intercept only model to 1586.8. Therefore, the new baseline model in the second block is **ermsoft** $\sim$ **ersandp**.[18]

Turning to the second block, again all possible extensions and their AIC values are listed. Because **ersandp** is already added, there is one possibility less but still it seems better to add another variable as the inclusion of **rterm** increases the AIC further to 1580.8. In the third block, **dinflation** is added to the baseline model, before in the last block the best possibility to extend the model is <**none**>. At this point, the algorithm stops and prints the coefficients of the last baseline model, which in this case is **ermsoft** $\sim$ **ersandp** + **rterm** + **dinflation**.

[18]To be more specific, this could also be written as **ermsoft** $\sim$ **1** + **ersandp**.

8 Quantile Regression

Reading: Brooks (2019, Section 4.10)

To illustrate how to run quantile regressions using R, we will now employ the simple CAPM beta estimation conducted in a previous section. We **re-open the 'capm.Rdata'** workfile. To estimate a quantile regression, the **quantreg** package needs to be installed and loaded, hence type **install.packages("quantreg")** into the *Console* or install the package using the menus. Do not forget to load the library afterwards using **library(quantreg)**.

This package provides the function **rq()** to estimate a quantile regression. The arguments are similar to the ones for **lm()** plus you need to specify the quantile that should be estimated in the argument, **tau**. The default setting for **tau** is 0.5, which is the median, but any value between 0 and 1 can be chosen.[19]. For now, we will not specify tau further and take a look at the median estimation. As before, we save the results in a new object and report them using the summary function. Hence, we type

```
qreg = rq(erford ~ ersandp, data = capm)
summary(qreg)
```

Pressing **Enter** should produce the following results.

```
> qreg = rq(erford ~ ersandp, data = capm)
> summary(qreg)

Call: rq(formula = erford ~ ersandp, data = capm)

tau: [1] 0.5

Coefficients:
            coefficients lower bd upper bd
(Intercept) -1.48968     -2.37032 -0.54627
ersandp      1.43846      1.06695  1.83852
>
```

Besides the coefficients for α and β, the results also provide confidence intervals for the estimated parameters by inverting a rank test, as described in Koenker (1994).[20]

While this command only provides estimates for one particular quantile, we might be interested in differences in the estimates across quantiles. Next, we generate estimates for a set of quantiles. Fortunately, the function **rq()** is implemented in such a way that it can vectorise the arguments for **tau**. This is not always the case, but very useful as we can just replace the single entry 0.5 from before. Note that we did not actually type **tau = 0.5** as it is implicit. Instead, we type

```
rq(erford ~ ersandp, data = capm, tau=seq(0.1,0.9,0.1))
```

This command makes use of the function **seq()** to create a sequence from a given start point to an end point using a defined step size. In this case, it starts at 0.1 and ends at 0.9 using steps of size 0.1. Hence, we obtain the estimations for all deciles. Pressing **Enter** provides the following results:

[19] Note that R will not produce an error for an input of **tau = 2**. However, you will not find any meaningful results.

[20] Specify the additional argument **se** of the summary function to receive results for standard errors. Possible specifications include **"iid"**, **"nid"**, **"ker"** and **"boot"** – see the help function for more detailed descriptions.

```
> rq(erford ~ ersandp, data = capm, tau=seq(0.1,0.9,0.1))
Call:
rq(formula = erford ~ ersandp, tau = seq(0.1, 0.9, 0.1), data = capm)

Coefficients:
             tau= 0.1  tau= 0.2  tau= 0.3  tau= 0.4  tau= 0.5    tau= 0.6 tau= 0.7 tau= 0.8  tau= 0.9
(Intercept) -11.92905 -7.349600 -4.878333 -3.347896 -1.489678 -0.01884444 1.756783 4.001337 10.456271
ersandp       2.34042  1.804414  1.659638  1.500533  1.438457  1.29670556 1.528341 1.619863  1.847333

Degrees of freedom: 193 total; 191 residual
>
```

For each decile, two estimates are presented: the β-coefficient on 'ersandp' and the intercept. Take some time to examine and compare the coefficient estimates across quantiles. What do you observe? We find a monotonic rise in the intercept coefficients as the quantiles increase. This is to be expected since the data on y have been arranged that way. But the slope estimates are very revealing - they show that the beta estimate is much higher in the lower tail than in the rest of the distribution of ordered data. Thus the relationship between excess returns on Ford stock and those of the S&P500 is much stronger when Ford share prices are falling most sharply. This is worrying, for it shows that the 'tail systematic risk' of the stock is greater than for the distribution as a whole. This is related to the observation that when stock prices fall, they tend to all fall at the same time, and thus the benefits of diversification that would be expected from examining only a standard regression of y on x could be much overstated.

An interesting diagnostic test is to see whether the coefficients for each quantile are significantly different from the OLS regression. The **quantreg** package has implemented this test such that it is easily accessible. It is part of the graphical output of the summary function

```
qreg = rq(erford ~ ersandp, data = capm, tau=seq(0.1,0.9,0.1))
plot(summary(qreg), level = 0.95)
```

Running the above code will produce the two graphs in Figure 9, showing the regression coefficients of the nine quantile regressions in black and their 95% confidence intervals in the gray shaded area. Further, the coefficient of the OLS regression is included in red with the 95% confidence band, the exact values were $\alpha_{OLS} = -0.956$ and $\beta_{OLS} = 1.890$. The graph clearly shows the increasing pattern in intercepts discussed in the paragraph above, and hence they are also not likely to be equal to the OLS intercept. For the β, however, most quantile estimations are within the 95% confidence interval, except for the 10, 40, 50 and 60% quantiles.

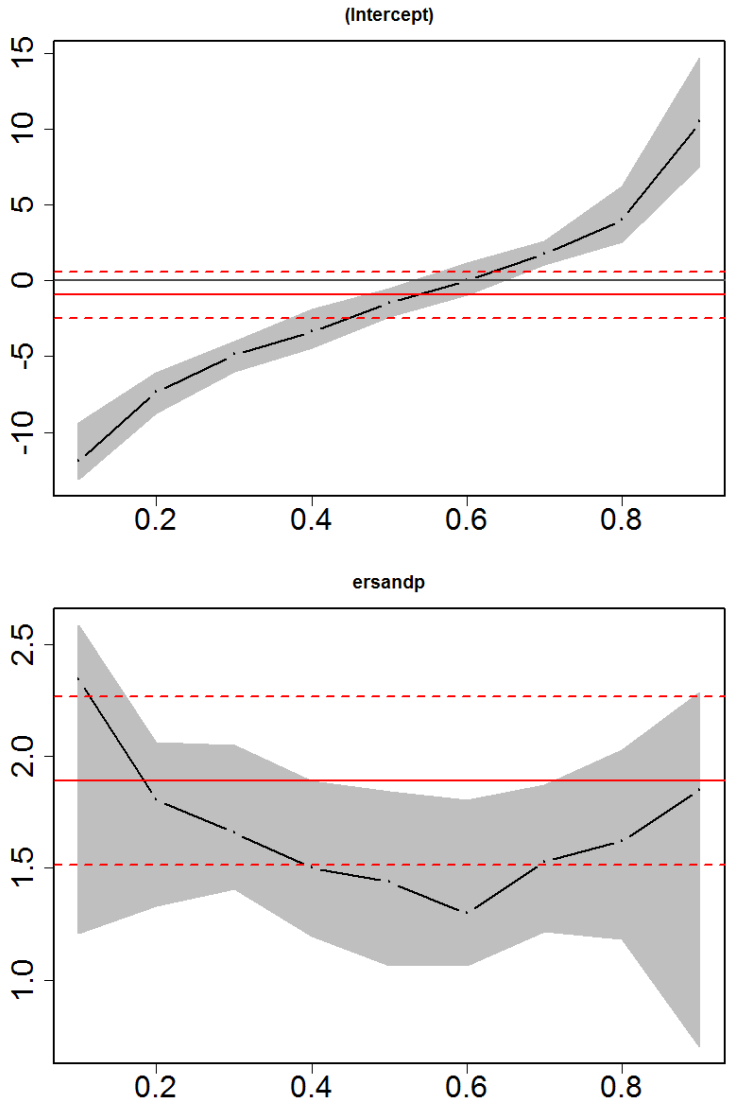


Figure 9: Plot of Coefficients from Quantile Regression Against OLS

9 Calculating Principal Components

Reading: Brooks (2019, Appendix 4.2)

In this section, we will examine a set of interest rates of different maturities and calculate the principal components for this set of variables. We can import the Treasury bill data directly from the Excel file **fred.xls** and save the dataset into **fred.RData**. To check whether the data was imported correctly, you can run **summary** on the six interest rates **GS3M, GS6M, GS1, GS3, GS5** and **GS10** and should obtain the following output.

```
> summary(fred[c("GS3M","GS6M","GS1","GS3","GS5","GS10")])
      GS3M            GS6M            GS1             GS3             GS5             GS10      
 Min.   : 0.010   Min.   : 0.040   Min.   : 0.100   Min.   : 0.330   Min.   : 0.620   Min.   : 1.500  
 1st Qu.: 0.940   1st Qu.: 1.030   1st Qu.: 1.230   1st Qu.: 1.760   1st Qu.: 2.480   1st Qu.: 3.570  
 Median : 4.320   Median : 4.490   Median : 4.520   Median : 4.740   Median : 5.070   Median : 5.340  
 Mean   : 3.981   Mean   : 4.168   Mean   : 4.343   Mean   : 4.937   Mean   : 5.318   Mean   : 5.821  
 3rd Qu.: 5.860   3rd Qu.: 6.190   3rd Qu.: 6.270   3rd Qu.: 7.040   3rd Qu.: 7.400   3rd Qu.: 7.710  
 Max.   :14.280   Max.   :14.810   Max.   :14.730   Max.   :14.730   Max.   :14.650   Max.   :14.590  
> 
```

To run a principal component analysis (PCA), the **stats** package provides the function **prcomp()**, which only needs the dataset as input. However, also set the argument **scale.** to **T** (true) to scale the variables to unit variance before rotating them.

```
> pca = prcomp(fred[c("GS3M","GS6M","GS1","GS3","GS5","GS10")],scale. = T)
> summary(pca)
Importance of components:
                          PC1     PC2     PC3     PC4     PC5      PC6
Standard deviation     2.4278 0.31481 0.07205 0.03745 0.01418 0.009422
Proportion of Variance 0.9823 0.01652 0.00087 0.00023 0.00003 0.000010
Cumulative Proportion  0.9823 0.99885 0.99972 0.99995 0.99999 1.000000
> pca$rotation
            PC1        PC2         PC3        PC4        PC5        PC6
GS3M -0.4071221  0.4634033 -0.52780134 -0.4857975  0.2786857  0.1651357
GS6M -0.4086842  0.3932419 -0.07368993  0.2948432 -0.5815993 -0.4977123
GS1  -0.4102502  0.2647232  0.34013817  0.5499158  0.2488373  0.5306553
GS3  -0.4112542 -0.1200790  0.54468637 -0.2900928  0.4243304 -0.5055272
GS5  -0.4090370 -0.3665644  0.21155894 -0.4126476 -0.5548485  0.4188402
GS10 -0.4030910 -0.6416785 -0.50878398  0.3467406  0.1850494 -0.1115026
> 
```

The function **prcomp()** returns a list of five variables, the standard deviations **sdev**, the transformation matrix **rotation**, the centring parameter of the transformation **center** and the scaling parameter **scale**. With the optional binary argument **retx**, you can also obtain the transformed variables, hence the scaling, centring and rotating applied to the original data series. Applying the summary function returns a matrix containing the standard deviations of the six principal components and their proportion of the total variance, as well as the cumulative proportion of all components up to the specific column. Typing **fred$rotation** will show the six eigenvectors or principal components.

It is evident that there is a great deal of common variation in the series, since the first principal component captures over 98% of the variation in the series and the first two components capture 99.9%. Consequently, if we wished, we could reduce the dimensionality of the system by using two components

rather than the entire six interest rate series. Interestingly, the first component comprises almost exactly equal weights in all six series while the second component puts a larger negative weight on the shortest yield and gradually increasing weights thereafter. This ties in with the common belief that the first component captures the level of interest rates, the second component captures the slope of the term structure (and the third component captures curvature in the yield curve).

10 Diagnostic Testing

10.1 Testing for Heteroscedasticity

Reading: Brooks (2019, Section 5.4)

In this example we will undertake a test for heteroscedasticity, using the **'macro.RData'** workfile. We will inspect the residuals of the APT-style regression of the excess return of Microsoft shares, 'ermsoft', on unexpected changes in a set of financial and macroeconomic variables, which we have estimated above. Thus, the first step is load the data set using **load("macro.RData")**.[21]

To get a first impression of the properties of the residuals, we want to plot them. The linear regression model **lm_msoft** has saved the residuals in the variable **residuals**, hence plotting is straight-forward by typing

```
plot(macro$Date[-(1:2)],lm_msoft$residuals,type = "l",xlab="",ylab="")
```

We need to take care to exclude the first two observations from the date variable, as they are not included in the regression. Forgetting to do so will result in an error as the lengths of the two series are not equal. The resulting plot should resemble Figure 10.

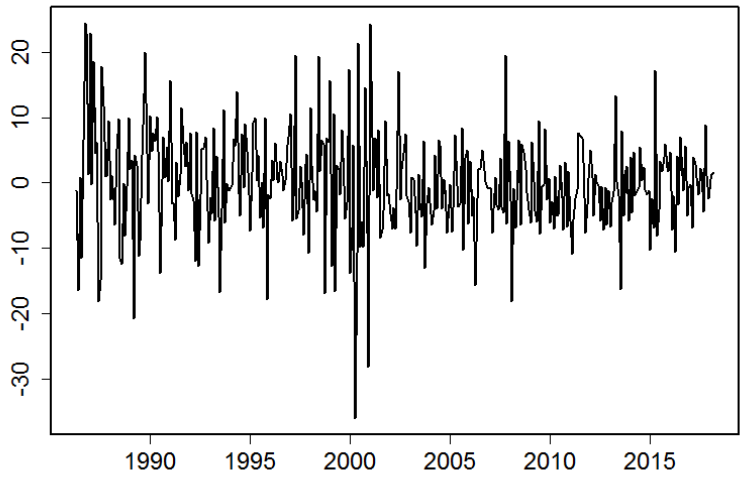


Figure 10: Plot of Residuals from Linear Regression

Let us examine the pattern of residuals over time. If the residuals of the regression have systematically changing variability over the sample, that is a sign of heteroscedasticity. In this case, it is hard to see any clear pattern (although it is interesting to note the considerable reduction in volatility post-2003), so we need to run the formal statistical test.

To do so, we install the package **lmtest** which includes tests, data sets, and examples for diagnostic checking in linear regression models. The two test demonstrated here are the Breusch–Pagan and the studentized Breusch–Pagan test. Both can be executed with the function **bptest()** from the **lmtest** package. The function only needs the regression formula and dataset as inputs. However, to run the

[21] Note this command only works if you have set your working directory to the path the file is stored in. Otherwise input the complete path.

(original) Bresuch–Pagan test, which assumes that the regression disturbances are normally distributed, the optional argument **studentize** has to be set to **F** (false), while with the default setting, **studentize = T**, the studentized Breusch–Pagan test is executed that drops the normality assumption. Hence, run the following two commands to obtain the output below.

```
bptest(formula(lm_msoft),data = macro,studentize = FALSE)
bptest(formula(lm_msoft),data = macro,studentize = TRUE)
```

```
> bptest(formula(lm_msoft),data = macro,studentize = F)

        Breusch-Pagan test

data:  formula(lm_msoft)
BP = 6.3131, df = 7, p-value = 0.5037

> bptest(formula(lm_msoft),data = macro,studentize = T)

        studentized Breusch-Pagan test

data:  formula(lm_msoft)
BP = 3.1607, df = 7, p-value = 0.8698

>
```

As can be seen, the null hypothesis is one of constant variance, i.e., homoscedasticity. With a χ^2-value of 6.31 and a corresponding p-value of 0.5037, the Breusch–Pagan test suggests not to reject the null hypothesis of constant variance of the residuals. This result is robust to the alternative distributional assumption, since the test statistic and p-value also suggest that there is not a problem of heteroscedastic errors for the APT-style model.

10.2 Using White's Modified Standard Error Estimates

Reading: Brooks (2019, Section 5.4.3)

The aim of this paragraph is to obtain heteroscedasticity-robust standard errors for the previous regression. Another way to reproduce the test statistics from the simple OLS regression is to use the function **coeftest** which is part of the package **lmtest**, that was already used in the previous subsection. The advantage of **coeftest** is that it offers the argument **vcov.** to specify the covariance matrix of the estimated coefficients and therefore to adjust the standard errors. The package **sandwich** provides several functions that automatically compute these covariance matrices. Therefore, after installing and loading the **sandwich** package, use the function **vcovHC()** for heteroskedasticity-consistent estimation of the covariance matrix.

The function **vcovHC()** is then called within the c**coeftest** function in the following way:

```
coeftest(lm_msoft,vcov. = vcovHC(lm_msoft,type="HC1"))
```

where the argument **type** specifies the standard errors. To obtain the estimator suggested by White (1980),use **"HC0"**. The possible alternatives "HC1", "HC2" and "HC3" are related to "HC0", but corrected for a factor of $\frac{n}{n-k}$, $\frac{1}{1-h}$ or $\frac{1}{(1-h)^2}$, respectively. The output below shows White standard errors with an adjustment for the degree of freedom k (**"HC1"**).

```
> coeftest(lm_msoft,vcov. = vcovHC(lm_msoft,type="HC1"))

t test of coefficients:

              Estimate Std. Error t value  Pr(>|t|)
(Intercept)  1.3260025  0.4590678  2.8885  0.004096 **
ersandp      1.2807988  0.0929615 13.7777 < 2.2e-16 ***
dprod       -0.3030317  0.6345495 -0.4776  0.633246
dcredit     -0.0253637  0.0208151 -1.2185  0.223790
dinflation   2.1946700  1.3068027  1.6794  0.093903 .
dmoney      -0.0068714  0.0109006 -0.6304  0.528835
dspread      2.2600645  3.4278130  0.6593  0.510088
rterm        4.7330689  1.7265468  2.7413  0.006412 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

>
```

Comparing the regression output for our APT-style model using robust standard errors with that using ordinary standard errors (section 7), we find that the changes in significance are only marginal. Of course, only the standard errors have changed and the parameter estimates remain identical to those estimated before. However, this has not resulted in changes about the conclusions reached about the significance or otherwise of any variable.

10.3 The Newey–West Procedure for Estimating Standard Errors

Reading: Brooks (2019, Section 5.5.7)

In this subsection, we will estimate the Newey–West heteroscedasticity and autocorrelation robust standard errors. Compared to the previous command, the only aspect that needs to be changed is the covariance matrix given by the argument **vcov.** in the **coeftest** function. Instead of **vcovHC**, use the function **NeweyWest** also provided by the **sandwich** package.

```
coeftest(lm_msoft,vcov. = NeweyWest(lm_msoft,lag = 6,adjust=T,prewhite=F))
```

As arguments, specify that R should use six lags by setting **lag** to 6. To disable the pre-whitening of functions, set **prewhite** to false and as before adjust for the degree of freedom by setting **adjust** to true. Press **Enter** and you should obtain the following results.

```
> coeftest(lm_msoft,vcov. = NeweyWest(lm_msoft,lag = 6,adjust = T,prewhite = F))

t test of coefficients:

              Estimate Std. Error t value  Pr(>|t|)
(Intercept)  1.3260025  0.5027048  2.6377  0.008694 **
ersandp      1.2807988  0.0998922 12.8218 < 2.2e-16 ***
dprod       -0.3030317  0.5216170 -0.5809  0.561625
dcredit     -0.0253637  0.0223410 -1.1353  0.256975
dinflation   2.1946700  1.3136172  1.6707  0.095614 .
dmoney      -0.0068714  0.0109474 -0.6277  0.530596
dspread      2.2600645  2.8418132  0.7953  0.426948
rterm        4.7330689  1.7585143  2.6915  0.007431 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

>
```

10.4 Autocorrelation and Dynamic Models

Reading: Brooks (2019, Subsections 5.5.7–5.5.11)

The simplest test for autocorrelation is due to Durbin and Watson (1951) (DW) and is implemented in **dwtest()** from the **lmtest** package. It can be applied in a straightforward way to the regression model without any further specification to obtain the following output.

```
> dwtest(lm_msoft)

        Durbin-Watson test

data:  lm_msoft
DW = 2.0974, p-value = 0.8176
alternative hypothesis: true autocorrelation is greater than 0

>
```

The value of the DW statistic is 2.10. What is the appropriate conclusion regarding the presence or otherwise of first order autocorrelation in this case? An alternative test for autocorrelation is the Breusch–Godfrey test. It is a more general test for autocorrelation than DW and allows us to test for higher order autocorrelation. It is also implemented in the **lmtest** package within the function **bgtest**. Again, executing is simple, the only argument to change is **order** which is 1 by default to allow for more lags to be included. Set **order** to 10, run the command and the results should appear as below.

```
> bgtest(lm_msoft,order = 10)

        Breusch-Godfrey test for serial correlation of order up to 10

data:  lm_msoft
LM test = 4.7666, df = 10, p-value = 0.9062

>
```

10.5 Testing for Non-Normality

Reading: Brooks (2019, Section 5.7)

This section looks at higher moments of the residual distribution to test the assumption of normality. The package **moments** includes functions to compute these moments and tests for normality. Assume that we would like to test whether the normality assumption is satisfied for the residuals of the APT-style regression of Microsoft stock on the unexpected changes in the financial and economic factors. Before calculating the actual test statistic, it might be useful to have a look at the data as this might give us a first idea as to whether the residuals might be normally distributed. The two functions **skewness()** and **kurtosis()** compute the third and fourth moments of the residuals as

```
> skewness(lm_msoft$residuals)
[1] -0.005612677
> kurtosis(lm_msoft$residuals)
[1] 4.994826
>
```

If the residuals follow a normal distribution, we expect a histogram of the residuals to be bell-shaped (with no outliers). To create a histogram of the residuals, we type

```
hist(lm_msoft$residuals,main = "")
box()
```

to generate the histogram in Figure 11. We removed the title by setting **main** to an empty string and added a box around the histogram using **box()**.

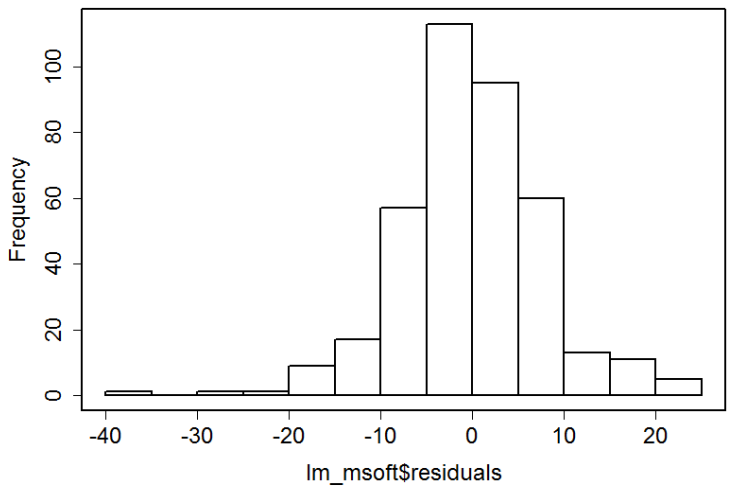


Figure 11: Histogram of Residuals

Looking at the histogram plot, we see that the distribution of the residuals roughly assembles a bell-shape, although we also find that there are some large negative outliers which might lead to a considerable negative skewness of the data series. We could increase the argument **breaks** to get a more differentiated histogram.

However, if we want to test the normality assumption of the residuals more formally it is best to turn to a formal normality test. One of the most commonly applied tests for normality is the Jarque–Bera (JB) test.[22] It is implemented in the function **jarque.test()** of the **moments** package. Further, there is the skewness test of D'Agostino (1970) and the kurtosis test of Anscombe and Glynn (1983), which are variations of the Jarque–Bera test. While the Jarque–Bera test is based on the skewness and kurtosis of a data series, the other functions test the two moments separately. The application of the tests is fairly easy, as they only need one argument, the residuals. Pressing **Enter** should provide the output below.

```
> jarque.test(lm_msoft$residuals)

        Jarque-Bera Normality Test

data:  lm_msoft$residuals
JB = 63.505, p-value = 1.621e-14
alternative hypothesis: greater

> agostino.test(lm_msoft$residuals)

        D'Agostino skewness test

data:  lm_msoft$residuals
skew = -0.0056127, z = -0.0456820, p-value = 0.9636
alternative hypothesis: data have a skewness

> anscombe.test(lm_msoft$residuals)

        Anscombe-Glynn kurtosis test

data:  lm_msoft$residuals
kurt = 4.9948, z = 4.5984, p-value = 4.258e-06
alternative hypothesis: kurtosis is not equal to 3

>
```

The results support the earlier observation that while the skewness does not diverge from that of a normal distribution (0), the kurtosis is significantly higher than that of a normal distribution (3). Hence, the D'Agostino test does not reject the null hypothesis of zero skewness, but the Anscombe–Glynn test rejects the null hypothesis of a kurtosis of 3. The joint Jarque–Bera test also rejects the normality assumption.

What could cause this strong deviation from normality? Having another look at the histogram, it appears to have been caused by a small number of very large negative residuals representing monthly stock price falls of more than 20%. The following subsection discusses a possible solution to this issue.

10.6 Dummy Variable Construction and Application

Reading: Brooks (2019, Subsection 5.7.2)

As we saw from the plot of the distribution above, the non-normality in the residuals from the Microsoft regression appears to have been caused by a small number of outliers in the sample. Such events can be identified if they are present by plotting the actual values and the residuals of the regression. We have created a plot containing the residuals of the Microsoft regression. Let us now add the fitted values. Fortunately, these are already saved in the variable **fitted.values** of the linear regression model. Thus,

[22] For more information on the intuition behind the Jarque–Bera test, please refer to section 5.7 in Brooks (2019).

we only need to add the second plot using the function **lines()**. To make them easier to distinguish, we also change the colour **(col)** and thickness **(lwd)**.

```
plot(macro$Date[-(1:2)],lm_msoft$residuals,type = "l", col="red",xlab="",
   ylab="")
lines(macro$Date[-(1:2)],lm_msoft$fitted.values)
legend("bottomright",c("Residuals","Fitted"), col = c("red","black"),lty=1)
```

These two lines of code will then produce the graph in Figure 12.

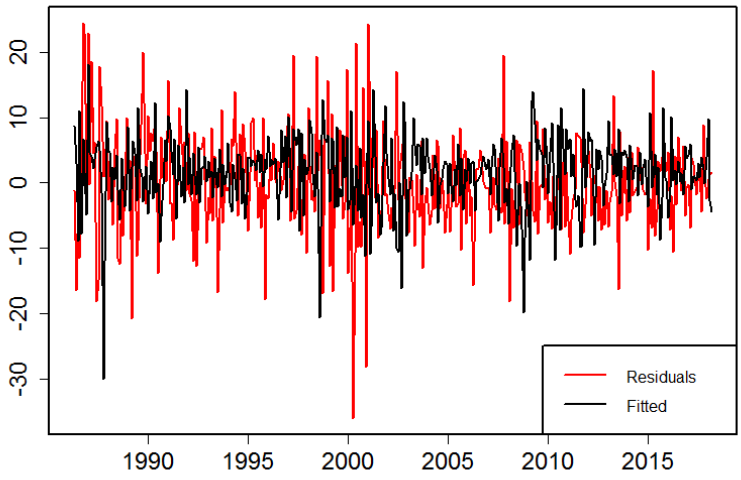


Figure 12: Regression Residuals and Fitted Series

From the graph, it can be seen that there are several large (negative) outliers, but the largest of all occur in 2000. All of the large outliers correspond to months where the actual return was much smaller (i.e., more negative) than the model would have predicted, resulting in a large residual. Interestingly, the residual in October 1987 is not quite so prominent because even though the stock price fell, the market index value fell as well, so that the stock price fall was at least in part predicted.

In order to identify the exact dates that the biggest outliers were realised, it is probably easiest to just examine a table of values for the residuals, by sorting the variable. The function **sort** applied to the residuals reveals that the 170^{th} and 178^{th} entries are the smallest, which represent the residuals in April (-36.075) and December 2000 (-28.143), respectively.

One way of removing the (distorting) effect of big outliers in the data is by using dummy variables. It would be tempting, but incorrect, to construct one dummy variable that takes the value 1 for both April 2000 and December 2000, but this would not have the desired effect of setting both residuals to zero. Instead, to remove two outliers requires us to construct two separate dummy variables. In order to create the Apr 00 dummy first, it is useful to change the format of the Date variable which is **POSIXct** to a simple **Date** type. After that, the dates can be easily compared using the '==' operator, such that the dummy variable can be constructed as a logical expression.

```
macro$Date = as.Date(macro$Date)
macro$APR00DUM = as.integer(macro$Date == as.Date("2000-04-01"))
macro$DEC00DUM = as.integer(macro$Date == as.Date("2000-12-01"))
```

The above code makes use of the functions **as.integer** and **as.Date** to cast variables into a specific format. These are inherited from the general class *object* and hence are available for any data format. The second (third) line transforms the logical expression in the parentheses into an integer of 0 or 1. The new variable takes the value 1 if the Date is equal to "2004-04-01" ("2004-12-01") and 0 otherwise.

Let us now rerun the regression to see whether the results change once we remove the effect of the two largest outliers. For this, we just add the two dummy variables APR00DUM and DEC00DUM to the list of independent variables and create the new linear regression model **lm_dummy**. The output of this regression should look as follows.

```
> lm_dummy = lm(ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney+
+                 dspread + rterm + APR00DUM + DEC00DUM, data = macro)
> summary(lm_dummy)

Call:
lm(formula = ermsoft ~ ersandp + dprod + dcredit + dinflation +
    dmoney + dspread + rterm + APR00DUM + DEC00DUM, data = macro)

Residuals:
     Min       1Q   Median       3Q      Max
-20.9130  -4.3820  -0.4323   4.2993  25.0430

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)    1.419760   0.454388   3.125 0.001920 **
ersandp        1.253853   0.090222  13.897  < 2e-16 ***
dprod         -0.321131   0.704945  -0.456 0.648985
dcredit       -0.015718   0.026044  -0.603 0.546548
dinflation     1.442064   1.214637   1.187 0.235889
dmoney        -0.005696   0.014868  -0.383 0.701875
dspread        1.869289   3.955274   0.473 0.636770
rterm          4.264174   1.640514   2.599 0.009712 **
APR00DUMTRUE -37.028834   7.576018  -4.888 1.52e-06 ***
DEC00DUMTRUE -28.729950   7.546383  -3.807 0.000164 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.492 on 373 degrees of freedom
  (2 observations deleted due to missingness)
Multiple R-squared:  0.406,     Adjusted R-squared:  0.3917
F-statistic: 28.33 on 9 and 373 DF,  p-value: < 2.2e-16

>
```

Note that the dummy variable parameters are both highly significant and take approximately the values that the corresponding residuals would have taken if the dummy variables had not been included in the model.[23] By comparing the results with those of the regression above that excluded the dummy variables, it can be seen that the coefficient estimates on the remaining variables change quite a bit in this instance. The inflation parameter is now insignificant and the R^2 value has risen from 0.34 to 0.41 because of the perfect fit of the dummy variables to those two extreme outlying observations.

Finally, we can re-examine the normality test results of the residuals based on this new model specification by applying the functions to the residuals of **lm_dummy**. We see that now also the skewness test strongly rejects the normality assumption with a p-value of 0.0016. The residuals are still a long way from following a normal distribution, and the joint null hypothesis of normality is still strongly rejected, probably because there are still several very large outliers. While it would be possible

[23] Note the inexact correspondence between the values of the residuals and the values of the dummy variable parameters because two dummies are being used together; had we included only one dummy, the value of the dummy variable coefficient and that which the residual would have taken would be identical.

to continue to generate dummy variables, there is a limit to the extent to which it would be desirable to do so. With this particular regression, we are unlikely to be able to achieve a residual distribution that is close to normality without using an excessive number of dummy variables. As a rule of thumb, in a monthly sample with 381 observations, it is reasonable to include, perhaps, two or three dummy variables for outliers, but more would probably be excessive.

10.7 Multicollinearity

Reading: Brooks (2019, Section 5.8)

Let us assume that we would like to test for multicollinearity issues in the Microsoft regression ('macro.RData' workfile). To generate the correlation matrix, we use the function **cor()** of the **stats** package. We specify the six columns and exclude the first two rows from the data to discard 'NA' values.

```
cor(macro[-(1:2),c("dprod","dcredit","dinflation",
    "dmoney","dspread","rterm")])
```

After typing the above command and pressing **Enter**, the following correlation matrix appears in the *Console*.

```
> cor(macro[-(1:2),c("dprod","dcredit","dinflation","dmoney","dspread","rterm")])
                dprod      dcredit  dinflation       dmoney     dspread        rterm
dprod       1.00000000  0.094273354 -0.14355079 -0.052514358 -0.05275628 -0.043750669
dcredit     0.09427335  1.000000000 -0.02460369  0.150165099  0.06281801 -0.004029469
dinflation -0.14355079 -0.024603694  1.00000000 -0.093571291 -0.22710010  0.041606256
dmoney     -0.05251436  0.150165099 -0.09357129  1.000000000  0.17069868  0.003800624
dspread    -0.05275628  0.062818012 -0.22710010  0.170698675  1.00000000 -0.017622374
rterm      -0.04375067 -0.004029469  0.04160626  0.003800624 -0.01762237  1.000000000
>
```

Do the results indicate any significant correlations between the independent variables? In this particular case, the largest observed correlations (in absolute value) are 0.17 between the money supply and spread variables, and –0.23 between the spread and unexpected inflation. These figures are probably sufficiently small that they can reasonably be ignored.

10.8 The RESET Test for Functional Form

Reading: Brooks (2019, Section 5.9)

To conduct the RESET test for the Microsoft regression, **lmtest** provides the function **resettest()**. Passing the regression model and the argument **power** to define which powers of the dependent variables should be included, it is sufficient to run a RESET test in R.

```
resettest(lm_msoft,power = 2:4)
```

Pressing **Enter** will present the following results.

```
> resettest(lm_msoft,power = 2:4)

        RESET test

data:  lm_msoft
RESET = 0.99377, df1 = 3, df2 = 372, p-value = 0.3957

>
```

The F-statistic has three degrees of freedom, since $\hat{y}^2, \hat{y}^3$ and $\hat{y}^4$ are included in the auxiliary regressions. With an F-value of 0.9938 and a corresponding p-value of 0.3957, the RESET test result implies that we cannot reject the null hypothesis that the model has no omitted variables. In other words, we do not find strong evidence that the chosen linear functional form of the model is incorrect.

10.9 Stability Tests

Reading: Brooks (2019, Section 5.12)

There are two types of stability tests presented in this section: the classical Chow test based on the F-distribution and recursive regression or CUSUM tests.

The package **strucchange** provides a variety of tools for testing, monitoring and dating structural changes in (linear) regression models. After installing and loading this package, we will start with the classical Chow test for a known or unknown break date. The specific function for this test is **Fstats** and its use is straightforward. Using only the regression formula and the dataset, the test is run by typing

```
sbtest = Fstats(formula(lm_msoft),data = macro)
```

which returns a list of nine with the most important variable **Fstats.** However, by default the test does not include the first and last 15% percent of the data to avoid break dates at the very beginning or end of the data set. The threshold can be altered within the function using the argument **from**. This circumstance makes it necessary to discuss in a bit more detail how to interpret the results as the variable **Fstats** now has 270 entries. Let us assume that we want to test whether a breakpoint occurred in January 1996, which is roughly in the middle of the sample period. To see the test statistic in **Fstats** for a structural break at this date, the date needs to be translated into an index.

Remember that the data set includes 385 observations of which only 383 are used in the regression. Out of these 383, the first and last 15% are excluded, which corresponds to 57 observations on each side, since $383 \times 0.15 = 57.45$, which is floored to 57. Therefore, the first entry in the vector **Fstats** corresponds to the break date with index $(385 - 383) + 57 = 59$, which is January 1991. Checking the index of January 1996 in the dataset **macro**, can be done with the function **match**. Typing

```
jan96 = match(as.Date("1996-01-01"),macro$Date)
```

will return 119. Now, the corresponding F-statistic can be found in the entry $119-2-57 = 60$. Knowing that this F or Chow-statistic has an asymptotic chi-squared distribution with $k = 8$ degrees of freedom helps to calculate the corresponding p-value as

```
chow = sbtest$Fstats[jan96-2-57]
1-pchisq(chow,8)
```

Running the code will result in the following output.

```
> sbtest$Fstats[jan96-2-57]
[1] 15.32228
> 1-pchisq(chow,sbtest$nreg)
[1] 0.05317363
>
```

The output presents the statistics of a Wald test of whether the coefficients in the Microsoft regression vary between the two subperiods, i.e., before and after January 1996. The null hypothesis is one of no structural break. We find that the Chow statistic is 15.3223 and that the corresponding p-value is 0.0532. Thus, we can reject the null hypothesis that the parameters are constant across the two subsamples at the 10% level.

Often, the date when the structural break occurs is not known in advance. Therefore, the Chow statistic is computed for every possible breakpoint in the given range. Applying the function **sctest** from the **strucchange** package will then run a supremum Wald test to compare the maximum statistic with what could be expected under the null hypothesis of no break.

```
sctest(sbtest)
bp = which.max(sbtest$Fstats)+59
macro$Date[bp]
```

To get the actual date of the suggested break, compute the index of the maximal statistic using the function **which.max()** and add 59, as explained above. The output obtained is displayed below.

```
> sctest(sbtest)

        supF test

data:  sbtest
sup.F = 32.465, p-value = 0.002149

> bp = which.max(sbtest$Fstats)+59
> macro$Date[bp]
[1] "2001-05-01"
>
```

Again, the null hypothesis is one of no structural breaks. The test statistic and the corresponding p-value suggest that we can reject the null hypothesis that the coefficients are stable over time, confirming that our model has a structural break for any possible break date in the sample. The test suggests May 2001 as the break date.

10.10 Recursive Estimation

Another way of testing whether the parameters are stable with respect to any break dates is to use one of the tests based on recursive estimation. To do so, run a small loop to re-estimate the regression with an increasing dataset and save the β estimates for the excess return on the S&P500 ('ersandp'). A possible set of code to do so could like the one below.

```
1 beta = NULL
2 for (t in 20:nrow(macro)){
```

```
3   lr = summary(lm(formula(lm_msoft), data = macro[3:t,]))
4   beta = rbind(beta,lr$coefficients["ersandp",1:2])
5 }
```

In the first line, the variable in which to save the beta estimates and standard errors is initialised as 'NULL'. Afterwards, the for loop begins signalled by **for**. In the same line, the sequence for which the loop is to be repeated is defined in brackets, here for all values between 20 and the number of rows of the dataset 'macro'. The statements that will be repeated are surrounded by curly brackets and in this case contain the estimation of a linear regression model in line 3 and the writing of coefficients into the variable **beta** in line 4. While the dataset used comprises the observations 3 to 20 in the first execution of the loop, it will increase to 3:21 for the second time, and so on until all observations from 3 to 385 are used. After every estimation, a summary is written to the intermediate variable **lr**, in the second step, and the first and second entry of the second row (this is the β and the respective σ) are appended to the matrix **beta**.

```
x_axis = macro$Date[20:nrow(macro)]
plot(x_axis,beta[,1],type = "l",ylim = c(0,3),xlab="",ylab="Beta")
lines(x_axis,beta[,1]+2*beta[,2],lty="dashed")
lines(x_axis,beta[,1]-2*beta[,2],lty="dashed")
```

To plot the computed β estimates and their standard errors, the above code is sufficient using the known functions **plot** and **lines**. Note that we need to adjust the x-axis, saved in the variable **x_axis**, to only include observations 20 to 385 as we discarded the first observations. With the argument **lty**, you can change the line style, turning the confidence bands into dashed lines.

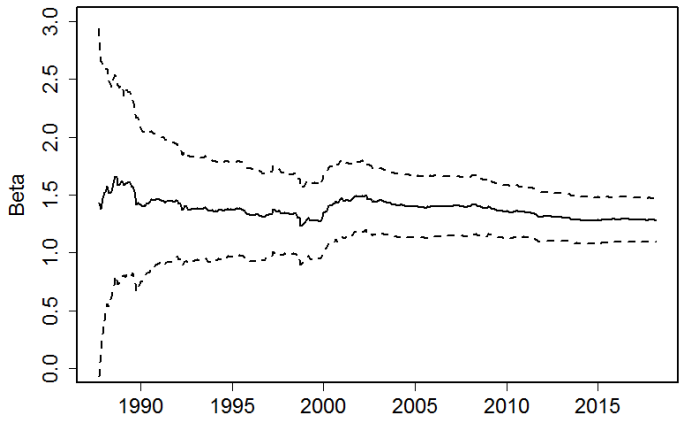


Figure 13: Plot of the Parameter Stability Test

What do we observe from the graph in Figure 13? The coefficients of the first couple of subsamples seem to be relatively unstable with large standard error bands while they seem to stabilise after a short period of time and only show small standard error bands. This pattern is to be expected, as it takes some time for the coefficients to stabilise since the first few sets are estimated using very small samples. Given this, the parameter estimates are remarkably stable. We can repeat this process for the recursive

estimates of the other variables to see whether they show similar stability over time.[24]

Another option is to run a so-called CUSUM test. The **strucchange** package provides the function **efp**; to run such a test and plot the results, type

```
plot(efp(lm_msoft,data=macro))
```

to obtain the graph in Figure 14

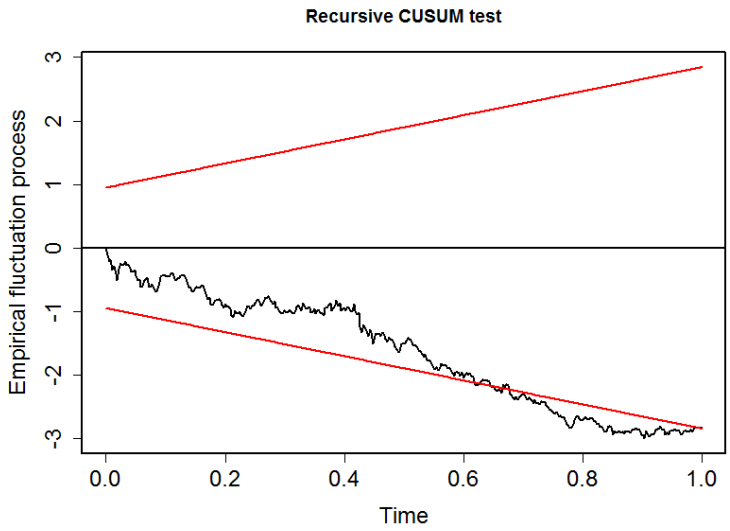


Figure 14: CUSUM Plot

Most of the time, the test statistic is within the 95% confidence bands, which means we cannot reject the null hypothesis of unstable parameters. However, the values are close to the boundary and in the period around 2010 we reject the null hypothesis.

[24]Note that to do this we only need to change the row specification in line 4 of the code from **"ersandp"** to the variable name we would like to inspect.

11 Constructing ARMA Models

Reading: Brooks (2019, Sections 6.4–6.7)

This example uses the monthly UK house price series which was already incorporated in section 2 (**'ukhp.RData'**). So first we re-load the workfile. There is a total of 326 monthly observations running from February 1991 (recall that the January observation is 'NA' because we are looking at returns) to March 2018 for the percentage change in house price series. Although in general it is not recommended to drop observations unless necessary, it will be more convenient to drop the first row of **UKHP** as the 'NA' value can cause trouble in several functions. To do so, execute the following line.

```
UKHP = UKHP[-1,]
```

The objective of this exercise is to build an ARMA model for the house price changes. Recall that there are three stages involved: identification, estimation and diagnostic checking. The first stage is carried out by looking at the autocorrelation and partial autocorrelation coefficients to identify any structure in the data.

11.1 Estimating Autocorrelation Coefficients

To generate autocorrelations and partial correlations, the **stats** package provides the functions **acf** and **pacf**. Setting the argument **lag.max** to twelve, you obtain a plot of the first 12 autocorrelations and partial autocorrelations as shown in Figure 15.

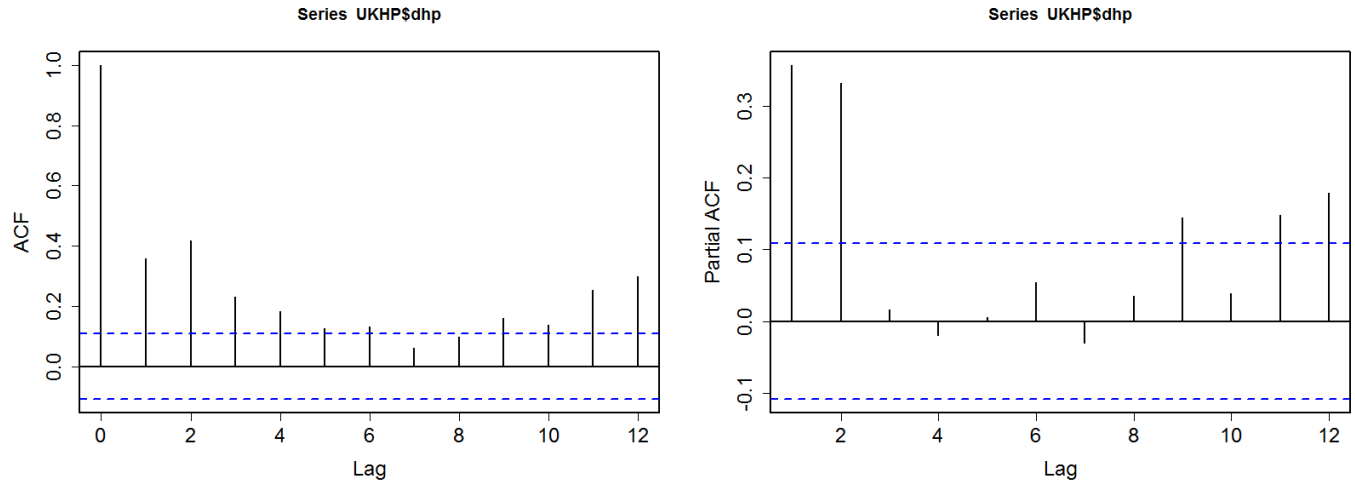


Figure 15: Autocorrelation and Partial Autocorrelation Functions

It is clearly evident that the series is quite persistent given that it is already in percentage change form: the autocorrelation function dies away rather slowly. Only the first two partial autocorrelation coefficients appear strongly significant.

Remember that as a rule of thumb, a given autocorrelation coefficient is classed as significant if it is outside a $\pm 1.96 \times 1/(T)^{1/2}$ band, where T is the number of observations. In this case, it would imply that a correlation coefficient is classed as significant if it is bigger than approximately 0.11 or smaller than –0.11. This band is also drawn in Figure 15 with blue colour. The band is of course wider when the sampling frequency is monthly, as it is here, rather than daily, where there would be more observations. It can be deduced that the first six autocorrelation coefficients (then nine through twelve) and the first

two partial autocorrelation coefficients (then nine, eleven and twelve) are significant under this rule. Since the first acf coefficient is highly significant, the joint test statistic presented in column 4 rejects the null hypothesis of no autocorrelation at the 1% level for all numbers of lags considered. It could be concluded that a mixed ARMA process might be appropriate, although it is hard to precisely determine the appropriate order given these results. In order to investigate this issue further, information criteria are now employed.

11.2 Using Information Criteria to Decide on Model Orders

An important point to note is that books and statistical packages often differ in their construction of the test statistic. For example, the formulae given in Brooks (2019) for Akaike's and Schwarz's Information Criteria are

$$\text{AIC} = \ln(\hat{\sigma}^2) + \frac{2k}{T} \tag{2}$$

$$\text{SBIC} = \ln(\hat{\sigma}^2) + \frac{k}{T}\ln(T) \tag{3}$$

where $\hat{\sigma}^2$ is the estimator of the variance of regressions disturbances u_t, k is the number of parameters and T is the sample size. When using the criterion based on the estimated standard errors, the model with the lowest value of AIC and SBIC should be chosen. The **AIC()** function of the **stats** package in R uses a formulation of the test statistic based on maximum likelihood estimation. The corresponding formulae are

$$\text{AIC}_L = -2 \cdot \ln(L) + 2 \cdot n \tag{4}$$

$$\text{SBIC}_L = -2 \cdot \ln(L) + \ln(T) \cdot n \tag{5}$$

where $\ln(L)$ is the maximised log-likelihood of the model. Unfortunately, this modification is not benign, since it affects the relative strength of the penalty term compared with the error variance, sometimes leading different packages to select different model orders for the same data and criterion!

Suppose that it is thought that ARMA models from order (0,0) to (5,5) are plausible for the house price changes. This would entail considering 36 models (ARMA(0,0), ARMA(1,0), ARMA(2,0), ..., ARMA(5,5)), i.e., up to 5 lags in both the autoregressive and moving average terms.

This can be done by separately estimating each of the models and noting down the value of the information criteria in each case. We can do this in the following way. Using the function **arima()** for an ARMA(1,1) model, we specify the argument **order** as **c(1,0,1)**. An equivalent formulation would be

$$\Delta HP_t = \mu + \varphi \Delta HP_{t-1} + \theta \epsilon_{t-1} + \epsilon_t, \tag{6}$$

with mean μ, AR coefficient φ and MA coefficient θ. We type

```
arima(UKHP$dhp,order = c(1,0,1))
```

into the *Console* and the following estimation output appears.

```
> arima(UKHP$dhp,order = c(1,0,1))

Call:
arima(x = UKHP$dhp, order = c(1, 0, 1))

Coefficients:
         ar1      ma1  intercept
      0.8244  -0.5441     0.4145
s.e.  0.0592   0.0875     0.1411

sigma^2 estimated as 0.9843:  log likelihood = -460.15,  aic = 928.3
>
```

In theory, the output would be discussed in a similar way to the simple linear regression model discussed in section 3. However, in reality it is very difficult to interpret the parameter estimates in the sense of, for example, saying 'a 1 unit increase in x leads to a β unit increase in y'. In part because the construction of ARMA models is not based on any economic or financial theory, it is often best not to even try to interpret the individual parameter estimates, but rather to examine the plausibility of the model as a whole, and to determine whether it describes the data well and produces accurate forecasts (if this is the objective of the exercise, which it often is).

In order to generate the information criteria corresponding to the ARMA(1,1) model, we use the function **AIC()**. Although the AIC criterion is already given in the output above, it can also be computed using this function. For the SBIC, the only change necessary is to set **k** to ln(**T**). Having saved the model from before into a new variable **ar11**, we apply the functions and obtain the output below.

```
> AIC(ar11)
[1] 928.2997
> AIC(ar11, k = log(nrow(UKHP)))
[1] 943.4473
>
```

We see that the AIC has a value of 928.30 and the BIC a value of 943.45. However, by themselves these two statistics are relatively meaningless for our decision as to which ARMA model to choose. Instead, we need to generate these statistics for the competing ARMA models and then select the model with the lowest information criterion. Repeating these steps for the other ARMA models would give all of the required values for the information criteria. The quickest way to this is within two loops running over the two variable the order of the AR part and the order of the MA part. A possible set of code could look like the one below.

```
1 aic_table = array(NA,c(6,6,2))
2 for (ar in 0:5) {
3   for (ma in 0:5) {
4     arma = arima(UKHP$dhp,order = c(ar,0,ma))
5     aic_table[ar+1,ma+1,1] = AIC(arma)
6     aic_table[ar+1,ma+1,2] = AIC(arma, k = log(nrow(UKHP)))
7   }
8 }
```

As this section is dealing with ARMA models and the way to choose the right order, the above code will not be discussed in detail. However, note that it makes use of the possibility to define three dimensional arrays in line 1. The table created by the code above is presented below with the AIC values in the first table and the SBIC values in the second table. Note that due to the fact that row and column indices start with 1, the respective AR and MA order represented is to be reduced by 1, i.e., entry (5,3) represents an ARMA(4,2) model.

```
> aic_table[,,1]
          [,1]     [,2]     [,3]     [,4]     [,5]     [,6]
[1,] 1001.2637 977.4020 935.5083 931.4545 930.2949 929.9505
[2,]  958.7914 933.4199 925.6487 923.7888 924.4999 929.3072
[3,]  922.4600 924.4080 926.2427 926.4088 926.1038 925.6015
[4,]  924.4016 926.4344 928.1834 925.9574 925.4296 918.6900
[5,]  926.2610 928.2587 914.0899 918.3987 927.4241 918.0929
[6,]  928.2454 927.6420 923.3094 917.8148 920.1052 927.6499
> which.min(aic_table[,,2])
[1] 3
> aic_table[,,1]
          [,1]     [,2]     [,3]     [,4]     [,5]     [,6]
[1,] 1001.2637 977.4020 935.5083 931.4545 930.2949 929.9505
[2,]  958.7914 933.4199 925.6487 923.7888 924.4999 929.3072
[3,]  922.4600 924.4080 926.2427 926.4088 926.1038 925.6015
[4,]  924.4016 926.4344 928.1834 925.9574 925.4296 918.6900
[5,]  926.2610 928.2587 914.0899 918.3987 927.4241 918.0929
[6,]  928.2454 927.6420 923.3094 917.8148 920.1052 927.6499
> which.min(aic_table[,,1])
[1] 17
> aic_table[,,2]
          [,1]     [,2]     [,3]     [,4]     [,5]     [,6]
[1,] 1008.8375 988.7627 950.6558 950.3889 953.0163 956.4588
[2,]  970.1521 948.5675 944.5832 946.5102 951.0082 959.6024
[3,]  937.6076 943.3425 948.9641 952.9171 956.3990 959.6836
[4,]  943.3361 949.1558 954.6917 956.2526 959.5117 956.5590
[5,]  948.9824 954.7670 944.3851 952.4808 965.2931 959.7488
[6,]  954.7536 957.9371 957.3914 955.6837 961.7611 973.0927
> which.min(aic_table[,,2])
[1] 3
>
```

So which model actually minimises the two information criteria? A quick way of finding the index of the minimal entry in a matrix is to use the function **which.min()**. Note that the index is not given in two dimensions, but as if the matrix would be a list of concatenated columns. In this case, the criteria choose different models: *AIC* selects an ARMA(4,2), while *SBIC* selects the smaller ARMA(2,0) model. It will always be the case that *SBIC* selects a model that is at least as small (i.e., with fewer or the same number of parameters) as *AIC*, because the former criterion has a stricter penalty term. This means that *SBIC* penalises the incorporation of additional terms more heavily. Many different models provide almost identical values of the information criteria, suggesting that the chosen models do not provide particularly sharp characterisations of the data and that a number of other specifications would fit the data almost as well.

12 Forecasting Using ARMA Models

Reading: Brooks (2019, Section 6.8)

Suppose that an AR(2) model selected for the house price percentage changes series was estimated using observations February 1991–December 2015, leaving 27 remaining observations to construct forecasts for and to test forecast accuracy (for the period January 2016–March 2018). Let us first estimate the ARMA(2,0) model for the time period from February 1991–December 2015. To account for this reduced dataset, simply put the restrictions into the square brackets for the dataset. The regression output is presented below.

```
> arima(UKHP$dhp[UKHP$Month <="2015-12-01"], order = c(2,0,0))

Call:
arima(x = UKHP$dhp[UKHP$Month <= "2015-12-01"], order = c(2, 0, 0))

Coefficients:
         ar1     ar2  intercept
      0.2355  0.3418     0.4266
s.e.  0.0542  0.0544     0.1364

sigma^2 estimated as 1.008:  log likelihood = -425.6,  aic = 859.2
>
```

Now that we have fitted the model, we can produce the forecasts for the period January 2016 to March 2018. There are two methods for constructing forecasts: dynamic and static. **Dynamic** forecasts are multi-step forecasts starting from the first period in the forecast sample. **Static** forecasts imply a sequence of one-step-ahead forecasts, rolling the sample forwards one observation after each forecast. In R, the simplest way to forecast is using the function **predict** which is part of the system library **stats**.[25] Although a static forecast only uses one-step ahead predictions, it is not as easy to implement, since the data has to be updated and **predict** does not allow for this. A dynamic forecast instead does not need any further input as it builds on the forecasted values to compute the next forecasts.

Thus, we can compute the static forecast directly after estimating the AR(2) model by typing

```
ar2 = arima(UKHP$dhp[UKHP$Month <="2015-12-01"], order = c(2,0,0))
dynamic_fc = predict(ar2,n.ahead = 27)
```

Since there are 27 out-of-sample observations, set the argument **n.ahead** to 27 to produce a 27-steps ahead forecast. For a one-step ahead forecast, **n.ahead** would only need to be altered to 1. But as the static forecast uses the out-of-sample observations, it is less straight-forward to compute. However, we can just use the formula for the one-step ahead forecast of AR models and update the data manually. Remember that for an AR(2) model like

$$\Delta HP_t = \mu + \varphi_1 \Delta HP_{t-1} + \varphi_2 \Delta HP_{t-2} + \epsilon_t, \tag{7}$$

the one-step ahead forecast is given by

$$\widehat{\Delta HP_{t+1}} = \mu + \varphi_1 \Delta HP_t + \varphi_2 \Delta HP_{t-1} + \epsilon_t, \tag{8}$$

[25] Note that there is also the package **forecast** which extends some forecasting techniques that are not necessary for this section.

and for any step h the static forecast is given as

$$\widehat{\Delta HP_{t+h}} = \mu + \varphi_1 \Delta HP_{t+h-1} + \varphi_2 \Delta HP_{t+h-2} + \epsilon_t, \tag{9}$$

Putting this into code is very efficiently done by typing

```
static_fc = ar2$coef[3]+ar2$coef[1]*UKHP$dhp[299:325]+ar2$coef[2]*UKHP$dhp[2
   98:324]
```

which again makes use of the way R interprets combinations of vectors and scalars. The variable **coef** contains the estimated coefficients φ_1, φ_2 and μ. Multiplied with the vectors of the past observations of **dhp**, the code resembles Equation (9) for h=1 to 27.
To spot differences between the two forecasts and to compare them to the actual values of the changes in house prices that were realised over this period, it is useful to create a graph of the three series. The code below is sufficient to do so. We have added some graphical options to show how to make graphs look more professional, while leaving the editing at a minimum.

```
par(lwd=2,cex.axis = 2)
plot(UKHP$Month[300:326],UKHP$dhp[300:326],type = "l",xlab = "",ylab = "")
lines(UKHP$Month[300:326],dynamic_fc$mean,col="blue")
lines(UKHP$Month[300:326],static_fc,col="red")
legend("topright", legend=c("Actual", "Dynamic", "Static"),col=c("black","
   blue","red"),lty= 1)
```

The resulting graph should resemble Figure 16.

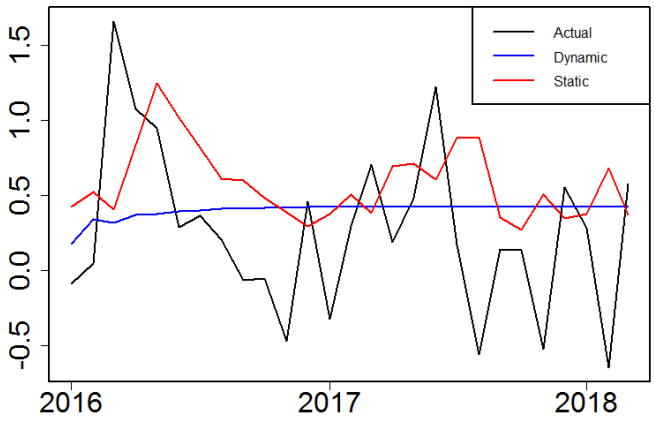


Figure 16: Graph Comparing Static and Dynamic Forecasts with the Actual Series

Let us have a closer look at the graph. For the dynamic forecasts, it is clearly evident that the forecasts quickly converge upon the long-term unconditional mean value as the horizon increases. Of course, this does not occur with the series of 1-step-ahead forecasts which seem to more closely resemble the actual 'dhp' series.

A robust forecasting exercise would of course employ a longer out-of-sample period than the two years or so used here, would perhaps employ several competing models in parallel, and would also compare the accuracy of the predictions by examining the forecast error measures, such as the square root of the mean squared error (RMSE), the MAE, the MAPE, and Theil's U-statistic.

13 Estimating Exponential Smoothing Models

Reading: Brooks (2019, Section 6.9)

Exponential smoothing models can be estimated in R using the package **smooth**. There is a variety of smoothing methods available, including single and double, or various methods to allow for seasonality and trends in the data. However, since single-exponential smoothing is the only smoothing method discussed in Brooks (2019), we will focus on this.

Using the same in-sample data as before and forecasting 27 steps ahead can be done using the function **es()** with the following input.

```
es(data = UKHP$dhp[1:299],h = 27)
```

Here we use the simpler notation for the observations before January 2016, as we now know these are the first 299 observations. The argument **h**, as before, defines the out-of-sample observations. After pressing **Enter**, the following output is obtained.

```
> es(data = UKHP$dhp[1:299],h = 27)
Time elapsed: 0.14 seconds
Model estimated: ETS(ANN)
Persistence vector g:
alpha
0.235
Initial values were optimised.
3 parameters were estimated in the process
Residuals standard deviation: 1.056
Cost function type: MSE; Cost function value: 1.103

Information criteria:
     AIC      AICc       BIC      BICc
883.8169 883.8983 894.9183 895.1501
>
```

The output includes the value of the estimated smoothing coefficient α (0.235 in this case), together with the root mean squared error (RMSE) or residual standard deviation for the whole forecast. The final in-sample smoothed value will be the forecast for those 27 observations (which in this case would be 0.2489906). You can find these forecasts by saving the forecast model to a variable, e.g., **smooth_fc** and query the value **forecast**.

14 Simultaneous Equations Modelling

Reading: Brooks (2019, Sections 7.5–7.9)

What is the relationship between inflation and stock returns? Clearly, they ought to be simultaneously related given that the rate of inflation will affect the discount rate applied to cashflows and therefore the value of equities, but the performance of the stock market may also affect consumer demand and therefore inflation through its impact on householder wealth (perceived or actual).

This simple example uses the same macroeconomic data as used previously **('macro.RData')** to estimate this relationship simultaneously. Suppose (without justification) that we wish to estimate the following model, which does not allow for dynamic effects or partial adjustments and does not distinguish between expected and unexpected inflation

$$inflation_t = \alpha_0 + \alpha_1 returns_t + \alpha_2 dcredit_t + \alpha_3 dprod_t + \alpha_4 dmoney + u_{1t} \quad (10)$$

$$returns_t = \beta_0 + \beta_1 dprod_t + \beta_2 dspread_t + \beta_3 inflation_t + \beta_4 rterm_t + u_{2t} \quad (11)$$

where 'returns' are stock returns – see Brooks (2019) for details.

It is evident that there is feedback between the two equations since the *inflation* variable appears in the *returns* equation and vice versa. Two-stage least squares (2SLS) is therefore the appropriate technique to use. To do this we need to specify a list of instruments, which would be all of the variables from the reduced form equation. In this case, the reduced form equations would be:

$$inflation = f(constant, dprod, dspread, rterm, dcredit, rterm, dmoney) \quad (12)$$

$$returns = g(constant, dprod, dspread, rterm, dcredit, rterm, dmoney) \quad (13)$$

For this example we will be using the 'macro.RData' file and the package **AER**, which provides the function **ivreg()** to run instrumental variable regressions. The function works in a similar fashion to the known **lm** function with the additional argument **instruments**. Hence, we put the exogenous variables on the right hand side of the formula and either define the instruments using the argument directly or after the exogenous variables separated by |. Also, as with other models, we can call them using the summary function to see the results in the following way.

```
> inf_iv = ivreg(inflation ~ rsandp + dprod + dcredit + dmoney |
+                  dcredit + dprod + rterm + dspread + dmoney, data = macro)
> summary(inf_iv)

Call:
ivreg(formula = inflation ~ rsandp + dprod + dcredit + dmoney |
    dcredit + dprod + rterm + dspread + dmoney, data = macro)

Residuals:
     Min       1Q   Median       3Q      Max
-1.40799 -0.33680 -0.01575  0.31978  2.61026

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.212932   0.037096   5.740 1.94e-08 ***
rsandp       0.103674   0.033564   3.089  0.00216 **
dprod        0.030860   0.050326   0.613  0.54012
dcredit     -0.005182   0.001917  -2.704  0.00717 **
dmoney      -0.002787   0.001062  -2.624  0.00905 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.5428 on 379 degrees of freedom
Multiple R-Squared: -1.827,     Adjusted R-squared: -1.857
Wald test: 5.375 on 4 and 379 DF,  p-value: 0.0003213

>
```

For the returns regression, we simply alter the formula and obtain the results below.

```
> ret_iv = ivreg(rsandp ~ inflation + dprod + dspread + rterm |
+                  dcredit + dprod + rterm + dspread + dmoney, data = macro)
> summary(ret_iv)

Call:
ivreg(formula = rsandp ~ inflation + dprod + dspread + rterm |
    dcredit + dprod + rterm + dspread + dmoney, data = macro)

Residuals:
     Min       1Q   Median       3Q      Max
-25.0829  -2.1293   0.2958   2.6311  11.9850

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.4802     0.6468   2.289  0.02266 *
inflation    -3.9592     2.8354  -1.396  0.16342
dprod        -0.1591     0.4045  -0.393  0.69437
dspread     -11.7333     3.7799  -3.104  0.00205 **
rterm        -0.3259     1.0564  -0.308  0.75791
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 4.391 on 379 degrees of freedom
Multiple R-Squared: -0.01033,   Adjusted R-squared: -0.02099
Wald test: 3.612 on 4 and 379 DF,  p-value: 0.006634

>
```

The results show that the stock index returns are a positive and significant determinant of inflation (changes in the money supply negatively affect inflation), while inflation has a negative effect on the

stock market, albeit not significantly so.

It may also be of relevance to conduct a Hausman test for the endogeneity of the inflation and stock return variables. To do this, we estimate the reduced form equations and add the fitted values to these equations. Hence, we run the simple OLS regressions and save the results as follows.

```
inf_ols = lm(inflation ~ dprod + dspread + rterm +  dcredit + dmoney, data =
    macro)
ret_ols = lm(rsandp ~ dprod + dspread + rterm +  dcredit + dmoney, data =
   macro)
macro$inffit = c(NA,inf_ols$fitted.values)
macro$retfit = c(NA,ret_ols$fitted.values)
```

Before we add the fitted values to the following two regressions.

```
> summary(lm(inflation ~ dprod + dcredit + dmoney + rsandp + retfit, data = macro))

Call:
lm(formula = inflation ~ dprod + dcredit + dmoney + rsandp +
    retfit, data = macro)

Residuals:
     Min       1Q   Median       3Q      Max
-1.68510 -0.15962 -0.01702  0.15680  0.99725

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.2129317  0.0202279  10.527  < 2e-16 ***
dprod        0.0308597  0.0274420   1.125    0.261
dcredit     -0.0051825  0.0010452  -4.958 1.08e-06 ***
dmoney      -0.0027873  0.0005793  -4.811 2.17e-06 ***
rsandp      -0.0025990  0.0035490  -0.732    0.464
retfit       0.1062734  0.0186425   5.701 2.41e-08 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.296 on 378 degrees of freedom
  (1 observation deleted due to missingness)
Multiple R-squared:  0.1616,    Adjusted R-squared:  0.1505
F-statistic: 14.57 on 5 and 378 DF,  p-value: 4.567e-13

>
```

```
> summary(lm(rsandp ~ dprod + dspread + rterm + inflation + inffit, data = macro))

Call:
lm(formula = rsandp ~ dprod + dspread + rterm + inflation + inffit,
    data = macro)

Residuals:
     Min       1Q   Median       3Q      Max
-24.8624  -2.3098   0.4615   2.6055  11.3393

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.4802     0.6313   2.345  0.01957 *
dprod        -0.1591     0.3948  -0.403  0.68728
dspread     -11.7333     3.6895  -3.180  0.00159 **
rterm        -0.3259     1.0312  -0.316  0.75217
inflation    -0.5708     0.7617  -0.749  0.45410
inffit       -3.3885     2.8705  -1.180  0.23856
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 4.286 on 378 degrees of freedom
  (1 observation deleted due to missingness)
Multiple R-squared:  0.03994,   Adjusted R-squared:  0.02724
F-statistic: 3.145 on 5 and 378 DF,  p-value: 0.008537

>
```

The conclusion is that the inflation fitted value term is not significant in the stock return equation and so inflation can be considered exogenous for stock returns. Thus it would be valid to simply estimate this equation (minus the fitted value term) on its own using OLS. But the fitted stock return term is significant in the inflation equation, suggesting that stock returns are endogenous.

15 Vector Autoregressive (VAR) Models

Reading: Brooks (2019, Section 7.10)

In this section, a VAR model is estimated in order to examine whether there are lead–lag relationships between the returns to three exchange rates against the US dollar – the euro, the British pound and the Japanese yen. The data are daily and run from 14 December 1998 to 3 July 2018, giving a total of 7,142 observations. The data are contained in the Excel file **'currencies.xls'**. First, we import the dataset into R and construct a set of continuously compounded percentage returns called **'reur', 'rgbp' and 'rjpy'** using the following set of commands, respectively

```
currencies$reur = c(NA,100*diff(log(currencies$EUR)))
currencies$rgbp = c(NA,100*diff(log(currencies$GBP)))
currencies$rjpy = c(NA,100*diff(log(currencies$JPY)))
currencies = currencies[-1,]
```

Note that we also remove the first observations to avoid problems with 'NA' values.

VAR estimation, diagnostic testing, forecasting and causality analysis in R is implemented in the package **vars**. To estimate a VAR(2) model for the currency returns, use the function **VAR** with input data and lag length.

```
VAR(currencies[c("reur","rgbp","rjpy")],p = 2)
```

The regression output from running the above code is shown on the next page. It is sectioned into the three dimensions of the vector.

```
> VAR(currencies[c("reur","rgbp","rjpy")],p = 2)

VAR Estimation Results:
=======================

Estimated coefficients for equation reur:
=========================================
Call:
reur = reur.l1 + rgbp.l1 + rjpy.l1 + reur.l2 + rgbp.l2 + rjpy.l2 + const

      reur.l1       rgbp.l1       rjpy.l1       reur.l2       rgbp.l2       rjpy.l2         const
 0.1474965506 -0.0183557359 -0.0070978299 -0.0118082131  0.0066228786 -0.0054272893  0.0001368541


Estimated coefficients for equation rgbp:
=========================================
Call:
rgbp = reur.l1 + rgbp.l1 + rjpy.l1 + reur.l2 + rgbp.l2 + rjpy.l2 + const

     reur.l1      rgbp.l1      rjpy.l1      reur.l2      rgbp.l2      rjpy.l2        const
-0.025270542  0.221361804 -0.039016040  0.046926637 -0.067794083  0.003286917  0.002825689


Estimated coefficients for equation rjpy:
=========================================
Call:
rjpy = reur.l1 + rgbp.l1 + rjpy.l1 + reur.l2 + rgbp.l2 + rjpy.l2 + const

      reur.l1       rgbp.l1       rjpy.l1       reur.l2       rgbp.l2       rjpy.l2         const
 0.0410612623 -0.0708456993  0.1324572148 -0.0188915075  0.0249075437  0.0149565492 -0.0004125908


>
```

We will shortly discuss the interpretation of the output, but the example so far has assumed that we know *the appropriate lag length* for the VAR. However, in practice, the first step in the construction of any VAR model, once the variables that will enter the VAR have been decided, will be to determine the appropriate lag length. For this purpose, **vars** provides the function **VARselect**, which again only needs the dataset and the maximal lag length to compute four information criteria. Applying the function to the dataset with a maximal lag length (**lag.max**) of 10 produces the following output.

```
> VARselect(currencies[c("reur","rgbp","rjpy")], lag.max = 10)
$selection
AIC(n)  HQ(n)  SC(n) FPE(n)
     4      2      1      4

$criteria
                1           2           3           4           5
AIC(n) -5.433453428 -5.43726462 -5.437666647 -5.438909241 -5.438464346
HQ(n)  -5.429472196 -5.43029747 -5.427713567 -5.425970237 -5.422539418
SC(n)  -5.421888924 -5.41702674 -5.408755386 -5.401324601 -5.392206329
FPE(n)  0.004367985  0.00435137  0.004349621  0.004344219  0.004346152
                6           7           8           9          10
AIC(n) -5.437389700 -5.436794113 -5.435808057 -5.435335655 -5.433668221
HQ(n)  -5.418478849 -5.414897337 -5.410925357 -5.407467031 -5.402813673
SC(n)  -5.382458304 -5.373189339 -5.363529904 -5.354384124 -5.344043311
FPE(n)  0.004350826  0.004353418  0.004357713  0.004359772  0.004367048

>
```

The results present the values of four information criteria: the Akaike (AIC), Hannan–Quinn (HQ), Schwarz (SC) and the Final Prediction Error (FPE) criteria. Under the variable **selection**, you can find the optimal lag length according to the four criteria. The AIC and FPE criteria select a lag length of four, while HQ suggests only two and SC chooses a VAR(1). Let us **estimate a VAR(1)** and examine the results. Does the model look as if it fits the data well? Why or why not?

We run the same command as before using a lag length of 1 and save the model into a new variable **var**.

```
var = VAR(currencies[c("reur","rgbp","rjpy")],p = 1)
```

Next, we run a Granger causality test which is implemented in the function **causality**. The input will be the estimated VAR model **var** and a string vector **cause** specifying from which variables the causality originates. This can be a single variable or a couple of variables. However, the function does not allow the user to specify the other side of the causality. Hence, it automatically performs a joint causality test on all remaining variables. This implies six possible tests with the summarised output below.

```
> causality(var,cause = c("reur","rgbp"))$Granger

        Granger causality H0: reur rgbp do not Granger-cause rjpy

data:  VAR object var
F-Test = 7.7511, df1 = 2, df2 = 21408, p-value = 0.0004315

> causality(var,cause = c("reur","rjpy"))$Granger

        Granger causality H0: reur rjpy do not Granger-cause rgbp

data:  VAR object var
F-Test = 7.7007, df1 = 2, df2 = 21408, p-value = 0.0004537

> causality(var,cause = c("rgbp","rjpy"))$Granger

        Granger causality H0: rgbp rjpy do not Granger-cause reur

data:  VAR object var
F-Test = 0.72376, df1 = 2, df2 = 21408, p-value = 0.4849

> causality(var,cause = "reur")$Granger

        Granger causality H0: reur do not Granger-cause rgbp rjpy

data:  VAR object var
F-Test = 4.1141, df1 = 2, df2 = 21408, p-value = 0.01635

> causality(var,cause = "rgbp")$Granger

        Granger causality H0: rgbp do not Granger-cause reur rjpy

data:  VAR object var
F-Test = 7.7523, df1 = 2, df2 = 21408, p-value = 0.000431

> causality(var,cause = "rjpy")$Granger

        Granger causality H0: rjpy do not Granger-cause reur rgbp

data:  VAR object var
F-Test = 7.5659, df1 = 2, df2 = 21408, p-value = 0.0005192

>
```

The null hypothesis of no Granger-causality is only rejected for the third case of GBP and JPY Granger-causing EUR. However, as mentioned above, these results are always based on the joint assumption of either two variables affecting one or one affecting two others.

To obtain the impulse responses for the estimated model, we use the function **irf** with input **var** and set the argument **n.ahead** to 20 to obtain a response for 20 steps ahead.

```
ir = irf(var,n.ahead = 20)
plot(ir)
```

Executing the above commands will produce the three plots in Figure 17.

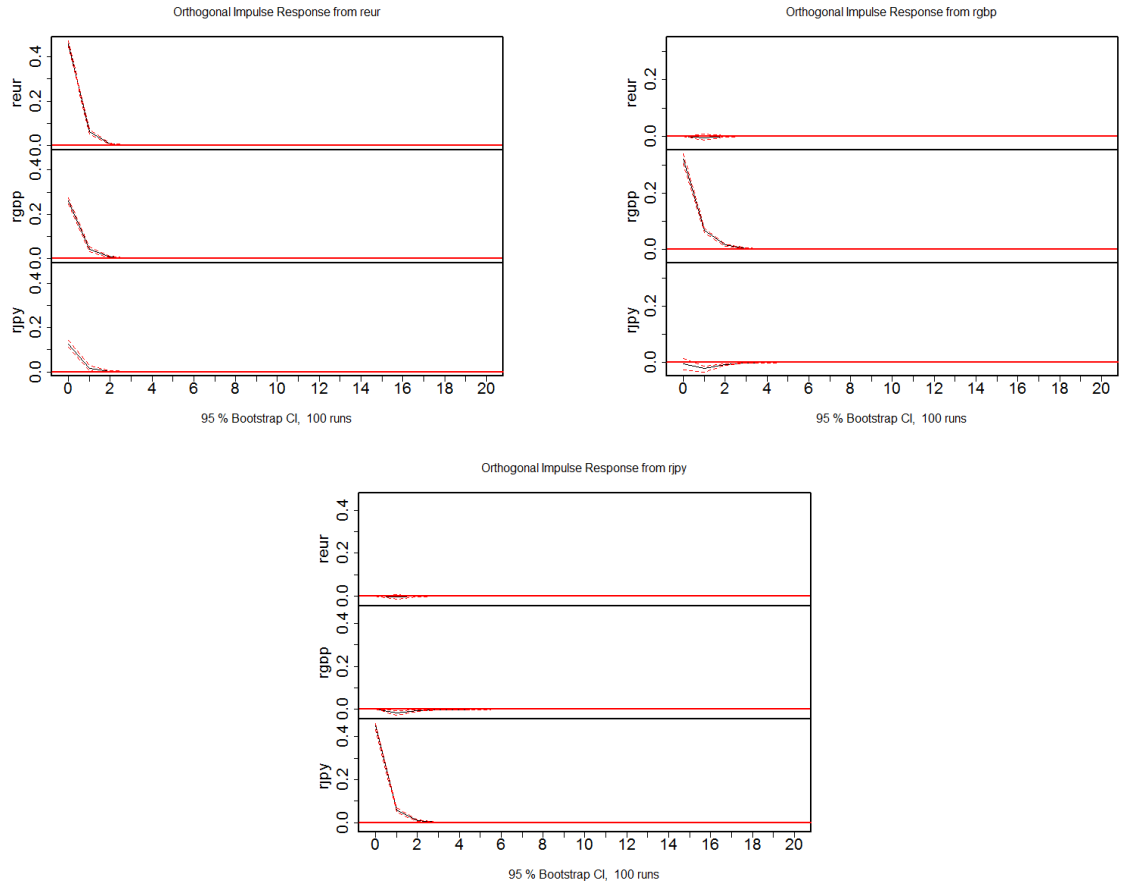


Figure 17: Graphs of Impulse Response Functions (IRFs) for the VAR(1) Model

As one would expect given the parameter estimates, only a few linkages between the series are established here. The responses to the shocks are very small, except for the response of a variable to its own shock, and they die down to almost nothing after the first lag.

To have a look at the forecast error variance decomposition, we can use the function **fevd**, with the same input, so we type

```
vd = fevd(var,n.ahead = 20)
plot(vd)
```

and receive the plot displayed in in Figure 18.

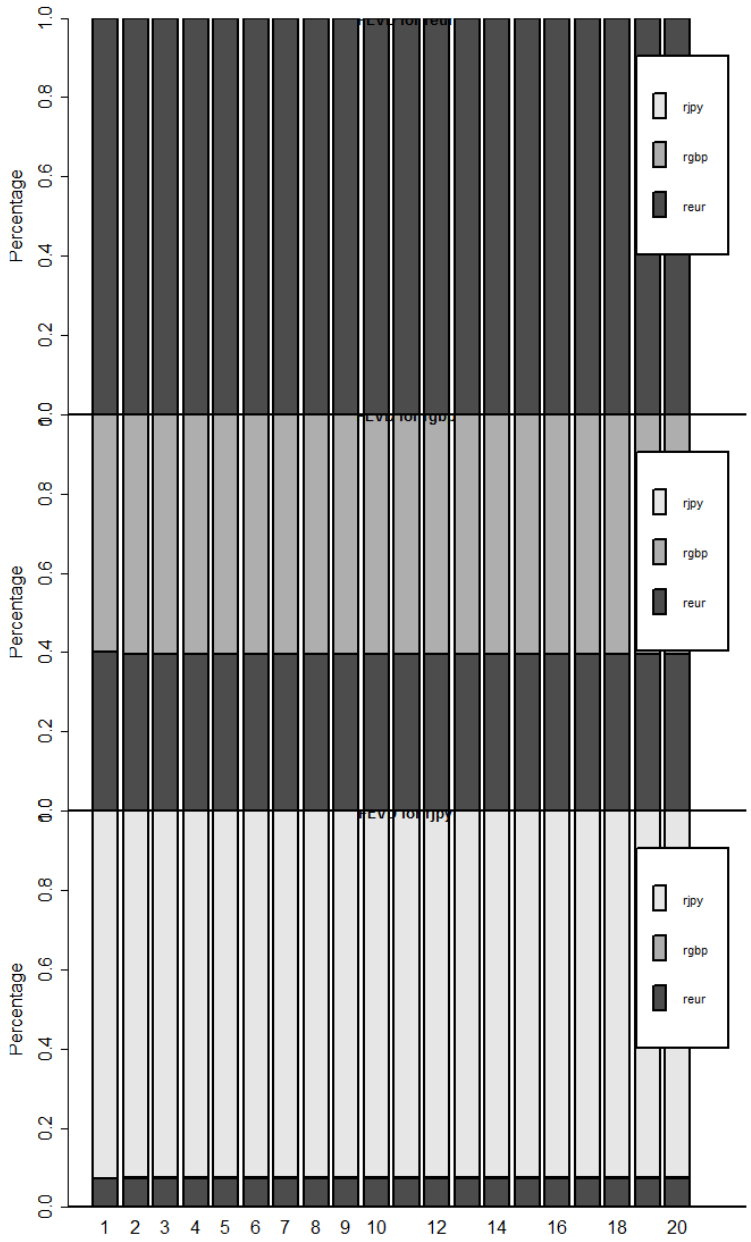


Figure 18: Graphs of FEVDs for the VAR(1) Model

There is again little that can be seen from these variance decomposition graphs apart from the fact that the behaviour is observed to settle down to a steady state very quickly. To illustrate how to interpret the FEVDs, let us have a look at the effect that a shock to the euro rate has on the other two rates and itself, which are shown in the first row of the FEVD plot. Interestingly, while the percentage of the errors that is attributable to own shocks is 100% in the case of the euro rate (top graph), for the pound, the euro series explains around 43% of the variation in returns (middle graph), and for the yen, the euro series explains around 7% of the variation.

We should remember that the ordering of the variables has an effect on the impulse responses and variance decompositions and when, as in this case, theory does not suggest an obvious ordering of the series, some sensitivity analysis should be undertaken. Let us assume we would like to test how sensitive the FEVDs are to a different way of ordering.

```
var_reverse = VAR(currencies[c("rjpy","rgbp","reur")],p = 1)
vd_reverse = fevd(var_reverse,n.ahead = 20)
plot(vd_reverse)
```

The above code is analogous to the steps we undertook before, with the small difference of arranging the vector of currencies in the reverse order **'rjpy'**, **'rgbp'** and **'reur'**. The output is presented in Figure 19. We can now compare the FEVDs of the reverse order with those of the previous ordering (Figure 18).

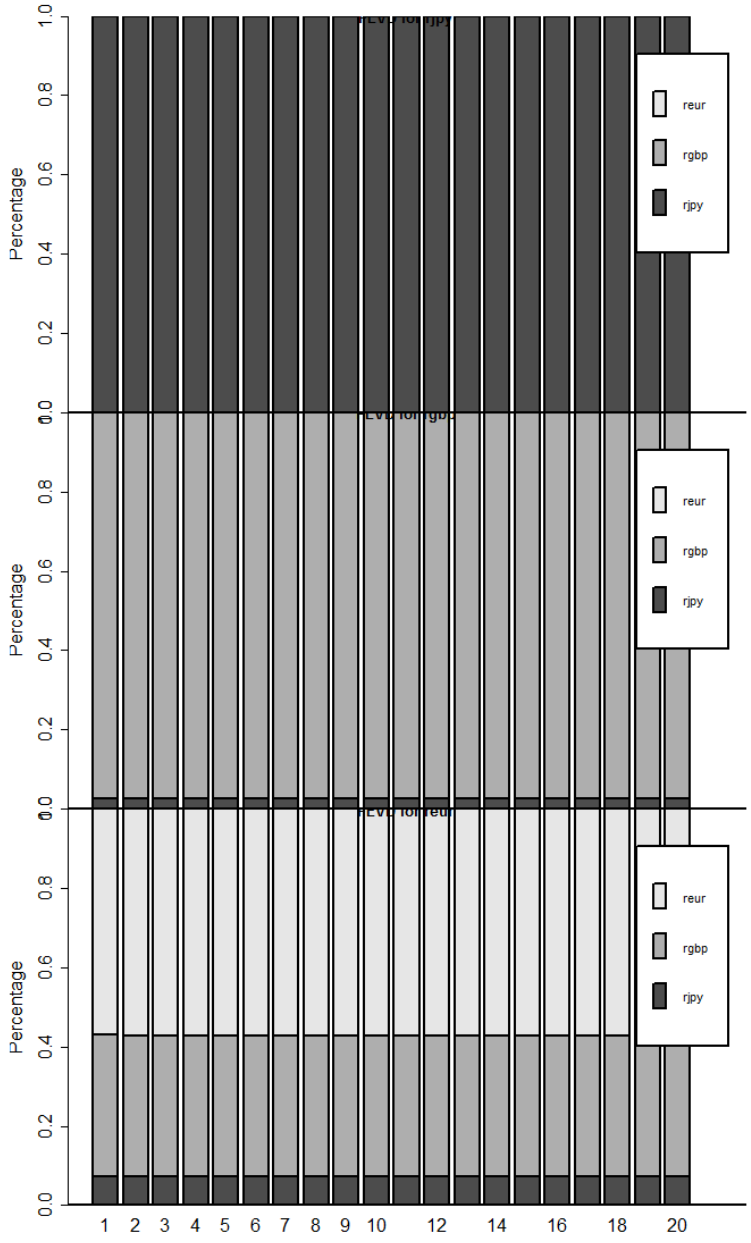


Figure 19: Graphs for FEVDs with Reverse Ordering

16 Testing for Unit Roots

Reading: Brooks (2019, Section 8.1)

In this section, we focus on how we can test whether a data series is stationary or not. This example uses the same data on UK house prices as employed previously (**'ukhp.RData'**). Assuming that the data have been loaded, and the variables are defined as before, we want to conduct a unit root test on the **hp** series. The package **fUnitRoots** provides various functions to test for unit roots, starting with the common Augmented Dickey–Fuller Test which is implemented in the function **adfTest**.

To run an ADF test in R, **adfTest** needs the data series as input, as well as a specification of the number of lags to be included. Apart from these essentials, you can specify the argument **type** as **nc** to not include a constant, **c** to include a constant and **ct** to include constant and trend. If we keep the default of including a constant, after running the code, we obtain the following output.

```
> adfTest(UKHP$hp,lags = 10,type = "c")

Title:
 Augmented Dickey-Fuller Test

Test Results:
  PARAMETER:
    Lag Order: 10
  STATISTIC:
    Dickey-Fuller: -0.4016
  P VALUE:
    0.903
```

The output shows the actual test statistics for the null hypothesis that the series 'hp' has a unit root. Clearly, the null hypothesis of a unit root in the house price series cannot be rejected with a p-value of 0.903.

Now we repeat all of the above step for the returns on the house price series **dhp**. The output would appear as below.

```
> adfTest(UKHP$dhp,lags = 10,type = "c")

Title:
 Augmented Dickey-Fuller Test

Test Results:
  PARAMETER:
    Lag Order: 10
  STATISTIC:
    Dickey-Fuller: -3.1081
  P VALUE:
    0.02778
```

We find that the null hypothesis of a unit root can be rejected for the returns on the house price series at the 5% level.[26] Alternatively, we can use a completely different test setting – for example, instead of the Dickey–Fuller test, we could run the Phillips–Perron test for stationarity. Among the options available, we only focus on one further unit root test that is strongly related to the Augmented Dickey-Fuller test presented above, namely the Dickey-Fuller GLS test (DF–GLS). Both test are implemented in the

[26] If we decrease the number of added lags, we find that the null hypothesis is rejected even at the 1% significance level.

function **ursersTest**.[27]

To run a DF–GLS test, we specify, the argument **type** to **"DF-GLS"**. This time, let us include a trend by setting **model** to **"trend"** rather than **"constant"**. Finally, set the maximum number of lags to 10. The following three lines summarise the code and results of the test.

```
adfgls = ursersTest(UKHP$hp, type = "DF-GLS", model = "trend", lag.max = 10)
adfgls@test$test@teststat
adfgls@test$test@cval
```

Note that **ursersTest** also produces graphical output of the test regression that will not be discussed here; it can be switched off by setting **doplot = F**. The second and third lines of the above code also introduce the operator which is similar to **$** and is a way to access variables. However, since the output of **ursersTest** is an object of a different class (not as usually a list), to address one of its attributes, also called slots, we need the **@** operator. The slot **test** is of list type and hence we can call its attribute test using **$** which is of a different class again and hence we need to call its attributes with **@**. The two attributes we look at are the test statistic **teststat** and the critical values **cval**.

```
> adfgls = ursersTest(UKHP$hp, type = "DF-GLS", model = "trend", lag.max = 10)
> adfgls@test[["test"]]@teststat
[1] -1.682445
> adfgls@test[["test"]]@cval
                 1pct  5pct 10pct
critical values -3.48 -2.89 -2.57
>
```

The conclusion, however, stays the same and the null hypothesis that the house price series has a unit root cannot be rejected. The critical values are obtained from Elliot et al. (1996).

[27] In fact, this function relies on the **urca** package for unit root test which we will study in the next section.

17 Cointegration Tests and Modelling Cointegrated Systems

Reading: Brooks (2019, Sections 8.3–8.11)

In this section, we will examine the S&P500 spot and futures series contained in the **'SandPhedge.RData'** workfile (that were discussed in section 3) for cointegration. We start with a test for cointegration based on the Engle–Granger approach where the residuals of a regression of the spot price on the futures price are examined. First, we **create two new variables**, for the log of the spot series and the log of the futures series, and call them **lspot** and **lfutures**, respectively. Then we run an OLS regression of **lspot** on **lfutures**:

```
SandPhedge$lspot = log(SandPhedge$Spot)
SandPhedge$lfutures = log(SandPhedge$Futures)
```

Note that is is not valid to examine anything other than the coefficient values in this regression, as the two series are non-stationary. Let us have a look at both the fitted and the residual series over time. As explained in previous sections, we find fitted values and residuals as variables of the new regression object **log_lm**. Generate a graph of the actual, fitted and residual series by first plotting only the **lspot** series and the fitted values and then setting the parameter **new=T**, before adding the plot of residuals. A possible set of code could look like this

```
1 par(lwd=2,cex.axis = 2)
2 plot(SandPhedge$Date,SandPhedge$lspot,type = "l",xlab = "",ylab = "",col="
    red")
3 lines(SandPhedge$Date,log_lm$fitted.values)
4 par(new=T)
5 plot(SandPhedge$Date,log_lm$residuals,col="blue",axes=F,type="l")
6 axis(side=4, at = pretty(range(log_lm$residuals)))
7 legend("bottomleft", legend=c("Actual", "Fitted"),col=c("black","red"),lty=
    1)
8 legend("bottomright", legend=c("Resid"),col=c("blue"),lty= 1)
```

Running the above code should produce the graph in Figure 20. Note that we have created a second y-axis for their values as the residuals are very small and we would not be able to observe their variation if they were plotted in the same scale as the actual and fitted values.

You will see a plot of the levels of the residuals (blue line), which looks much more like a stationary series than the original spot series (the red line corresponding to the actual values of y). Note how close together the actual and fitted lines are – the two are virtually indistinguishable and hence the very small right-hand scale for the residuals.

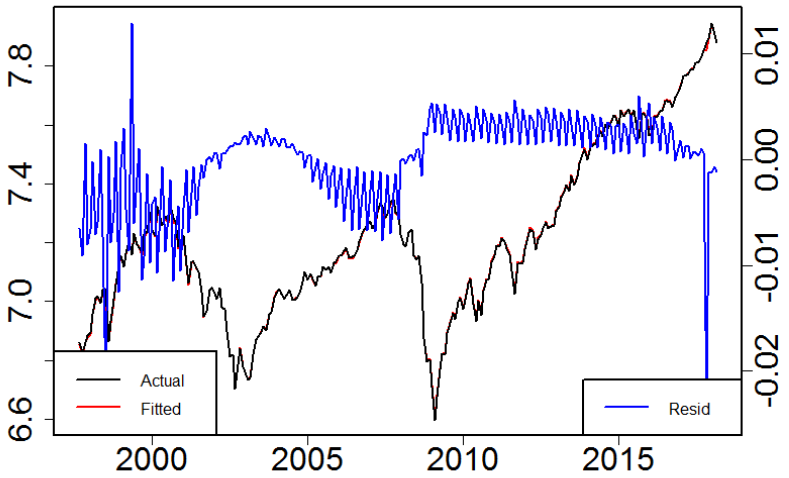


Figure 20: Actual, Fitted and Residual Plot

Let us now perform an DF–GLS Test on the residual series. Specifying the lag length as **12**, including a **trend**, we can directly look at the results as they appear below.

```
> urersTest(log_lm$residuals, type = "DF-GLS", model = "trend", lag.max = 12)@test$test@teststat
[1] -1.458417
> urersTest(log_lm$residuals, type = "DF-GLS", model = "trend", lag.max = 12)@test$test@cval
                 1pct  5pct 10pct
critical values -3.48 -2.89 -2.57
>
```

For twelve lags, we have a test statistic of (-1.458) which is not more negative than the critical values, even at the 10% level. Thus, the null hypothesis of a unit root in the test regression residuals cannot be rejected and we would conclude that the two series are not cointegrated. This means that the most appropriate form of the model to estimate would be one containing only first differences of the variables as they have no long-run relationship.

If instead we had found the two series to be cointegrated, an error correction model (ECM) could have been estimated, as there would be a linear combination of the spot and futures prices that would be stationary. The ECM would be the appropriate model in that case rather than a model in pure first difference form because it would enable us to capture the long-run relationship between the series as well as their short-run association. We could estimate an error correction model by running the following regression.

```
> summary(lm(SandPhedge$rspot[-1] ~ SandPhedge$rfutures[-1] + log_lm$residuals[-247]))

Call:
lm(formula = SandPhedge$rspot[-1] ~ SandPhedge$rfutures[-1] +
    log_lm$residuals[-247])

Residuals:
     Min       1Q   Median       3Q      Max
-2.44248 -0.06592  0.04480  0.17764  1.58732

Coefficients:
                        Estimate Std. Error t value Pr(>|t|)
(Intercept)             0.009331   0.025210   0.370    0.712
SandPhedge$rfutures[-1] 0.984771   0.005781 170.344   <2e-16 ***
log_lm$residuals[-247] -55.060243   5.784972  -9.518   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3936 on 243 degrees of freedom
Multiple R-squared:  0.9918,    Adjusted R-squared:  0.9917
F-statistic: 1.473e+04 on 2 and 243 DF,  p-value: < 2.2e-16

>
```

Note that the above model regresses the spot returns on futures returns and lagged residuals (the error correction term). While the coefficient on the error correction term shows the expected negative sign, indicating that if the difference between the logs of the spot and futures prices is positive in one period, the spot price will fall during the next period to restore equilibrium, and vice versa, the size of the coefficient is not really plausible as it would imply a large adjustment. Given that the two series are not cointegrated, the results of the ECM need to be interpreted with caution and a model that regresses spot returns on futures returns, lagged spot and lagged futures returns, would be more appropriate. Note that we can either include or exclude the lagged terms and both forms would be valid from the perspective that all of the elements in the equation are stationary.

Before moving on, we should note that this result is not an entirely stable one – for instance, if we run the regression containing no lags (i.e., the pure Dickey–Fuller test) or on a subsample of the data, we should find that the unit root null hypothesis should be rejected, indicating that the series are cointegrated. We thus need to be careful about drawing a firm conclusion in this case.

17.1 The Johansen Test for Cointegration

Although the Engle–Granger approach is evidently very easy to use, as outlined above, one of its major drawbacks is that it can estimate only up to one cointegrating relationship between the variables. In the spot–futures example, there can be at most one cointegrating relationship since there are only two variables in the system. But in other situations, if there are more variables, there can potentially be more than one linearly independent cointegrating relationship. Thus, it is appropriate instead to examine the issue of cointegration within the Johansen VAR framework. For this purpose, we will use the R package **urca**, which provides unit root and cointegration tests for time series data.

The application we will now examine centres on whether the yields on Treasury bills of different maturities are cointegrated. For this example we will use the **'fred.RData'** workfile which we created in section 9. It contains six interest rate series corresponding to 3 and 6 months, and 1, 3, 5, and 10 years. Each series has a name in the file starting with the letters 'GS'.

The first step in any cointegration analysis is to ensure that the variables are all non-stationary in their levels form, so **confirm that this is the case** for each of the six series, by running a unit root test on each one using either the **adfTest** function of the **fUnitRoots** package from the previous section or

the function **ur.ers** supplied by the **urca** package.[28]

Before specifying the VECM using the Johansen method, it is often very useful to graph the variables to see how they are behaving over time and with respect to each other. This will also help us to select the correct option for the VECM specification, e.g., if the series appears to follow a linear trend. To generate the graph, we just type the following lines

```
plot(fred$Date,fred$GS3M,type="l",xlab="",ylab="")
lines(fred$Date,fred$GS6M,col="red")
lines(fred$Date,fred$GS1,col="blue")
lines(fred$Date,fred$GS3,col="brown")
lines(fred$Date,fred$GS5,col="orange")
lines(fred$Date,fred$GS10,col="darkgreen")
```

and Figure 21 should appear. Note that we dropped the labels to make the legend easier to read. We can do this by setting the arguments **xlab** and **ylab** to empty strings "".

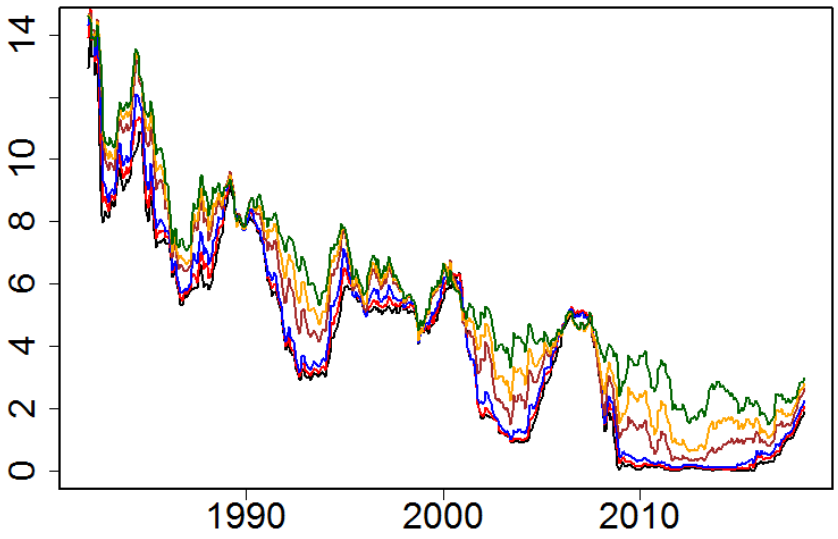


Figure 21: Graph of the Six US Treasury Interest Rates

We see that the series generally follow a linear downward trend, though some series show stronger inter-temporal variation, with large drops, than other series. Additionally, while all series seem to be related in some way, we find that the plots of some rates resemble each other more strictly than others, e.g., the GS3M, GS6M and GS1 rates.

To test for cointegration or fit cointegrating VECMs, we must also specify how many lags to include in the model. To select the optimal number of lags we can use the method shown in section 15. Hence, we apply the **VARselect** function form the **vars** package to receive the following output.

[28]The results are not presented here, but note that using 3 lags and a trend, for all series the null hypothesis of no unit root cannot be rejected.

```
> VARselect(fred[c("GS3M","GS6M","GS1","GS3","GS5","GS10")],lag.max = 12)
$selection
AIC(n)  HQ(n)  SC(n) FPE(n)
     2      2      2      2

$criteria
                1             2             3             4             5             6
AIC(n) -3.048588e+01 -3.088381e+01 -3.087091e+01 -3.087216e+01 -3.082098e+01 -3.083823e+01
HQ(n)  -3.032768e+01 -3.059002e+01 -3.044151e+01 -3.030717e+01 -3.012039e+01 -3.000205e+01
SC(n)  -3.008544e+01 -3.014014e+01 -2.978399e+01 -2.944201e+01 -2.904759e+01 -2.872161e+01
FPE(n)  5.756508e-14  3.867052e-14  3.918238e-14  3.915114e-14  4.123784e-14  4.057676e-14
                7             8             9            10            11            12
AIC(n) -3.081106e+01 -3.081602e+01 -3.074647e+01 -3.070735e+01 -3.065424e+01 -3.058976e+01
HQ(n)  -2.983928e+01 -2.970863e+01 -2.950349e+01 -2.932877e+01 -2.914007e+01 -2.893998e+01
SC(n)  -2.835121e+01 -2.801292e+01 -2.760014e+01 -2.721778e+01 -2.682144e+01 -2.641372e+01
FPE(n)  4.175760e-14  4.163494e-14  4.474874e-14  4.668409e-14  4.942441e-14  5.296642e-14

>
```

The four information criteria provide conclusive results regarding the optimal lag length. All four criteria, FPE, AIC, HQIC and SBIC, suggest an optimal lag length of two. Note, that this might not always be the case and it is then up to the reader to decide on the appropriate lag length. In the framework of this example, we follow the information criteria and select a lag length of two.

The next step of fitting a VECM is the actual Johansen procedure of determining the number of cointegrating relationships. The procedure works by comparing the log likelihood functions for a model that contains the cointegrating equation(s) and a model that does not. If the log likelihood of the unconstrained model that includes the cointegrating equations is significantly different from the log likelihood of the constrained model that does not include the cointegrating equations, we reject the null hypothesis of no cointegration. This is implemented in the function **ca.jo** of the **urca** package.

As formerly determined, use the six interest rates and a lag order (**K**) of two. The argument **ecdet** can be set to **"none"**, **"const"** or **"trend"**. We use **"const"**, since we want to allow for the cointegrating relationship to be stationary around a constant mean, but we do not want linear time trends in the levels of the data. Further, set the argument **type** to **"trace"** to obtain the trace statistic rather than the maximum eigenvalue statistic (**"eigen"**). Setting the values as described and applying the summary function to the output generates the results on the next page.

The first line summarises the test specifications produced and the six eigenvalues (λ) from which the test statistics are derived are presented. There then follows a table with the main results. The first column states the number of cointegrating relationships for the set of interest rates or the cointegrating rank r. In the second column, the test statistic, hence λ_{trace}, is displayed followed by the critical values at the 10%, 5% and 1% levels.

```
> summary(ca.jo(fred[c("GS3M","GS6M","GS1","GS3","GS5","GS10")],K=2,ecdet = "const",type = "trace"))

######################
# Johansen-Procedure #
######################

Test type: trace statistic , without linear trend and constant in cointegration

Eigenvalues (lambda):
[1] 1.933608e-01 1.441940e-01 1.112046e-01 4.191212e-02 1.951328e-02 1.837159e-02 2.121848e-17

Values of teststatistic and critical values of test:

           test 10pct   5pct   1pct
r <= 5 |   8.07  7.52   9.24  12.97
r <= 4 |  16.64 17.85  19.96  24.60
r <= 3 |  35.26 32.00  34.91  41.07
r <= 2 |  86.54 49.65  53.12  60.16
r <= 1 | 154.28 71.86  76.07  84.45
r = 0  | 247.75 97.18 102.14 111.01

Eigenvectors, normalised to first column:
(These are the cointegration relations)

             GS3M.12    GS6M.12     GS1.12     GS3.12     GS5.12    GS10.12    constant
GS3M.12    1.0000000  1.0000000  1.0000000    1.00000  1.0000000  1.0000000   1.0000000
GS6M.12   -1.8361728 -1.7240301 -7.8260887  -45.44358 -2.2414315 -2.1083618   0.7974764
GS1.12     0.6584909  1.5591332 10.3520558   64.84908  1.0119005  0.4455494  -2.7026864
GS3.12     0.7935402 -2.0124537 -9.3559469   36.99190  0.9399850  0.6478099   3.1335814
GS5.12    -0.7572572  1.4945981  8.5581204 -111.48938 -1.0083630 -7.7888536 -11.5201333
GS10.12    0.1275615 -0.2493571 -2.7878414   55.56632  0.4129490  8.1180175  11.7683970
constant   0.1581499 -0.1180595  0.5424579   -9.32844 -0.5161628 -5.7936204 -28.8343237

Weights W:
(This is the loading matrix)

          GS3M.12     GS6M.12      GS1.12        GS3.12      GS5.12       GS10.12      constant
GS3M.d 0.4401418 -0.27515704  0.03139674  0.0004933753 -0.04251249  0.0042441020 -6.309699e-17
GS6M.d 0.6134722 -0.12926923  0.01290303  0.0000976568 -0.03596756  0.0053450249 -1.122967e-16
GS1.d  0.5562986 -0.04534895 -0.01178610 -0.0001540930 -0.04416737  0.0059033018 -8.745449e-17
GS3.d  0.4869795  0.16241326 -0.01436364  0.0006761659 -0.05816293  0.0046972595 -2.045579e-16
GS5.d  0.5302487  0.18077756 -0.02830696  0.0011619122 -0.05582945  0.0025825107 -2.047967e-16
GS10.d 0.5536302  0.18370135 -0.02705497  0.0006235189 -0.05394235 -0.0001778005 -1.878066e-16

>
```

Reading the top panel of the table from bottom to top, the last row tests the null hypothesis of no cointegrating vectors against the alternative hypothesis that the number of cointegrating equations is strictly larger than the number assumed under the null hypothesis, i.e., larger than zero. The test statistic of 247.75 considerably exceeds the critical value (111.01) and so the null of no cointegrating vector is rejected. If we then move up to the next row, the test statistic (154.28) again exceeds the critical value so that the null of at most one cointegrating vector is also rejected. This continues, and we also reject the null of at most two cointegrating vectors, but we stop at the next row, where we do not reject the null hypothesis of at most three cointegrating vectors at the 1% level. If we use a significance level of 5%, we would still reject this hypothesis and would only stop at the next row.

Besides the λ_{trace} statistic, we can also employ an alternative statistic, the maximum-eigenvalue statistic (λ_{max}) by setting **type** to **"eigen"**. In contrast to the trace statistic, the maximum-eigenvalue statistic assumes a given number of r cointegrating relations under the null hypothesis and tests this against the alternative that there are $r + 1$ cointegrating equations.

The test output should now report the results for the λ_{max} statistics. We find that the results from the λ_{max} test confirm our previous results and show that we cannot reject the null hypothesis of three

cointegrating relations against the alternative of four cointegrating relations between the interest rates. It is therefore reasonable to assume a cointegration rank of three.

Now that we have determined the lag length, trend specification and the number of cointegrating relationships, we can fit the VECM model. To do so, we use the function **cajorls** from the **urca** package. This requires the output object of **ca.jo** and the cointegration rank **r=3** as an input and delivers the following results.

```
> vecm = ca.jo(fred[c("GS3M","GS6M","GS1","GS3","GS5","GS10")],K=2,ecdet = "const",type = "trace")
> cajorls(vecm,r=3)
$rlm

Call:
lm(formula = substitute(form1), data = data.mat)

Coefficients:
          GS3M.d      GS6M.d      GS1.d       GS3.d       GS5.d       GS10.d
ect1       0.1963815   0.4971060   0.4991636   0.6350292   0.6827193   0.7102766
ect2      -0.5795111  -1.0045571  -0.8510384  -1.0617728  -1.0637614  -1.1215328
ect3       0.1858437   0.3359907   0.1736022   0.4252023   0.3379851   0.3709008
GS3M.dl1   0.8046578   1.0052603   0.7937785   0.7297938   0.7851838   0.7296653
GS6M.dl1  -1.3002259  -1.7480493  -1.3158642  -1.4676407  -1.4515766  -1.3219279
GS1.dl1    0.7261699   1.0482154   0.7538492   0.9610662   0.7470956   0.6686458
GS3.dl1    0.3414507  -0.0154097  -0.0679247  -0.2322351  -0.0695104  -0.3048205
GS5.dl1   -0.1705721   0.2690776   0.4427480   0.5318636   0.3428756   0.4572949
GS10.dl1  -0.0816765  -0.1924225  -0.2043656  -0.1335948  -0.0008461   0.0573718


$beta
                  ect1          ect2          ect3
GS3M.l2   1.000000e+00  1.942890e-16  4.163336e-17
GS6M.l2  -2.953367e-16  1.000000e+00  2.520770e-18
GS1.l2    1.481020e-16  3.146355e-16  1.000000e+00
GS3.l2   -2.499094e+00 -2.786098e+00 -2.768639e+00
GS5.l2    1.573232e+00  2.073284e+00  2.242124e+00
GS10.l2   9.893932e-02 -1.585894e-01 -3.987532e-01
constant -5.346689e-01 -4.664695e-01 -2.485984e-01

>
```

R produces two tables as the main output. The first table contains the estimates of the short-run parameters. The three coefficients on 'ect1', 'ect2' and 'ect3' are the parameters in the adjustment matrix α for this model. The second table contains the estimated parameters of the cointegrating vectors for this model with a unit diagonal.

18 Volatility Modelling

In this section, we will use the '**currrencies.RData**' dataset. The exercises of this section will employ returns on daily exchange rates where there are 7,141 observations. Models of this kind are inevitably more data intensive than those based on simple linear regressions and hence, everything else being equal, they work better when the data are sampled daily rather than at a lower frequency.

18.1 Estimating GARCH Models

Reading: Brooks (2019, Section 9.9)

To estimate a GARCH-type model in R, we will use the package **rugarch**. Although it might seem a bit more complicated to become familiar with, it provides plenty of methods to estimate, forecast, simulate and plot different volatility models.[29] To estimate a GARCH model, the function **ugarchfit** is used. This requires a **uGARCHspec** object as input, which defines the lag orders, mean and variance equation. Hence, before we can estimate the model we create this object and save it by typing

```
spec = ugarchspec(mean.model = list(armaOrder=c(0,0)),variance.model = list(
   garchOrder=c(1,1),model="sGARCH"))
ugarchfit(spec,data=currencies$rjpy)
```

The two arguments specify the mean and variance model used, hence the mean equation will be an ARMA(0,0) model only containing the unconditional mean of the series, while the variance equation will follow a GARCH(1,1) process. The additional argument **model="sGARCH"** stands for standard GARCH, to distinguish different models of GARCH type. We do not include any further independent variables. To finally estimate the model, we need to provide the data to the function **ugarchfit** together with the **uGARCHspec** object we just created. This is done in the second line of the above code and results in the output on the next page.

Note that the actual output also includes some residual diagnostics that we do not display here. The top panel of the output again specifies the model estimated, before the estimated parameters are displayed. The coefficients on both the lagged squared residuals (**alpha1**) and lagged conditional variance term (**beta1**) are highly statistically significant. The other two coefficients **mu** and **omega** denote the intercept terms in mean and variance equation.

Also, as is typical of GARCH model estimates for financial asset returns data, the sum of the coefficients on the lagged squared error and lagged conditional variance is very close to unity (approximately 0.99). This implies that shocks to the conditional variance will be highly persistent. This can be seen by considering the equations for forecasting future values of the conditional variance using a GARCH model given in a subsequent section. A large sum of these coefficients will imply that a large positive or a large negative return will lead future forecasts of the variance to be high for a protracted period.

[29] The package **fGarch** provides similar methods, so the reader might also look into this package.

```
> ugarchfit(spec,data=currencies$rjpy)

*---------------------------------*
*          GARCH Model Fit        *
*---------------------------------*

Conditional Variance Dynamics
-----------------------------------
GARCH Model     : sGARCH(1,1)
Mean Model      : ARFIMA(0,0,0)
Distribution    : norm

Optimal Parameters
------------------------------------
        Estimate  Std. Error  t value Pr(>|t|)
mu      0.006403    0.004903   1.3059  0.19157
omega   0.001640    0.000216   7.6027  0.00000
alpha1  0.036849    0.001690  21.8038  0.00000
beta1   0.956418    0.001184 808.0878  0.00000

Robust Standard Errors:
        Estimate  Std. Error   t value Pr(>|t|)
mu      0.006403    0.005947    1.0768  0.28157
omega   0.001640    0.000608    2.6964  0.00701
alpha1  0.036849    0.003673   10.0336  0.00000
beta1   0.956418    0.000656 1457.4241  0.00000

LogLikelihood : -4344.782

Information Criteria
------------------------------------

Akaike       1.2180
Bayes        1.2218
Shibata      1.2180
Hannan-Quinn 1.2193
```

The individual conditional variance coefficients are also as one would expect. The variance intercept term _cons in the 'ARCH' panel is very small, and the 'ARCH'-parameter 'alpha1' is around 0.037 while the coefficient on the lagged conditional variance 'beta1' is larger at 0.956.

18.2 EGARCH and GJR Models

Reading: Brooks (2019, Sections 9.10–9.13)

Since the GARCH model was developed, numerous extensions and variants have been proposed. In this section we will estimate two of them, the EGARCH and GJR models. The GJR model is a simple extension of the GARCH model with an additional term added to account for possible asymmetries. The exponential GARCH (EGARCH) model extends the classical GARCH by correcting the non-negativity constraint and by allowing for asymmetries in volatility.

We start by estimating the EGARCH model. Within the **rugarch** package this can be done very easily – we only need to alter the argument **model** in the variance equation to **"eGARCH"**. Running the code again results in the following output.

```
> spec=ugarchspec(mean.model=list(armaOrder=c(0,0)),variance.model=list(garchOrder=c(1,1),model="eGARCH"))
> ugarchfit(spec,data=currencies$rjpy)

*---------------------------------*
*          GARCH Model Fit        *
*---------------------------------*

Conditional Variance Dynamics
-----------------------------------
GARCH Model     : eGARCH(1,1)
Mean Model      : ARFIMA(0,0,0)
Distribution    : norm

Optimal Parameters
------------------------------------
        Estimate  Std. Error   t value Pr(>|t|)
mu      0.003184    0.004752   0.66998  0.50287
omega  -0.011004    0.003005  -3.66223  0.00025
alpha1 -0.028417    0.001211 -23.45692  0.00000
beta1   0.986437    0.002677 368.49474  0.00000
gamma1  0.102905    0.007284  14.12805  0.00000

Robust Standard Errors:
        Estimate  Std. Error   t value Pr(>|t|)
mu      0.003184    0.006328   0.50315 0.614856
omega  -0.011004    0.009327  -1.17981 0.238077
alpha1 -0.028417    0.009388  -3.02708 0.002469
beta1   0.986437    0.007032 140.28573 0.000000
gamma1  0.102905    0.026866   3.83030 0.000128

LogLikelihood : -4323

Information Criteria
------------------------------------

Akaike       1.2122
Bayes        1.2170
Shibata      1.2122
Hannan-Quinn 1.2138
```

Looking at the results, we find that all EARCH and EGARCH terms are statistically significant. The EARCH terms represent the influence of news – lagged innovations – in the Nelson (1991) EGARCH model. The term 'alpha1' captures the

$$\frac{\upsilon_{t-1}}{\sqrt{\sigma^2_{t-1}}}$$

term and 'gamma1' captures the

$$\frac{|\upsilon_{t-1}|}{\sqrt{\sigma^2_{t-1}}} - \sqrt{\frac{2}{\pi}}$$

term. The negative estimate on the 'alpha1' term implies that negative shocks result in a lower next period conditional variance than positive shocks of the same sign. The result for the EGARCH asymmetry term is the opposite to what would have been expected in the case of the application of a GARCH model to a set of stock returns. But, arguably, neither the *leverage effect* or *volatility effect* explanations for asymmetries in the context of stocks apply here. For a positive return shock, the results suggest more yen per dollar and therefore a strengthening dollar and a weakening yen. Thus, the EGARCH results suggest that a strengthening dollar (weakening yen) leads to higher next period volatility than when the yen strengthens by the same amount.

Let us now test a GJR model. Again, the only aspect that needs to be changed is the argument **model** in the **uGARCHspec** object. After setting this to **"gjrGARCH"** and running the code, we obtain the following results. Note that part of the output is again suppressed.

```
> spec=ugarchspec(mean.model=list(armaOrder=c(0,0)),variance.model=list(garchOrder=c(1,1),model="gjrGARCH"))
> ugarchfit(spec,data=currencies$rjpy)

*---------------------------------*
*          GARCH Model Fit        *
*---------------------------------*

Conditional Variance Dynamics
-----------------------------------
GARCH Model     : gjrGARCH(1,1)
Mean Model      : ARFIMA(0,0,0)
Distribution    : norm

Optimal Parameters
------------------------------------
        Estimate  Std. Error   t value Pr(>|t|)
mu      0.003522    0.004913   0.71688 0.473446
omega   0.002037    0.000333   6.12304 0.000000
alpha1  0.027579    0.003291   8.37903 0.000000
beta1   0.950473    0.004008 237.11669 0.000000
gamma1  0.026850    0.005349   5.01943 0.000001

Robust Standard Errors:
        Estimate  Std. Error   t value Pr(>|t|)
mu      0.003522    0.005938   0.59321 0.553043
omega   0.002037    0.000764   2.66590 0.007678
alpha1  0.027579    0.006218   4.43531 0.000009
beta1   0.950473    0.006388 148.79058 0.000000
gamma1  0.026850    0.010414   2.57813 0.009934

LogLikelihood : -4329.648

Information Criteria
------------------------------------

Akaike       1.2140
Bayes        1.2188
Shibata      1.2140
Hannan-Quinn 1.2157
```

Similar to the EGARCH model, we find that all ARCH, GARCH and GJR terms are statistically significant. However it is important to recognise the definition of the model as it depends on how the dummy variable is constructed since it can alter the sign of coefficients. Here, the 'gamma1' term captures the $v_{t-1}^2 I_{t-1}$ term where $I_{t-1} = 1$ if $v_{t-1}^2 > 0$ and $I_{t-1} = 0$ otherwise. We find a positive coefficient estimate on the 'gamma1' term, which again is not what we would expect to find according to the *leverage effect* explanation if we were modelling stock return volatilities.[30]

18.3 GARCH-M Estimation

Reading: Brooks (2019, Section 9.15)

To estimate a GARCH-M model in R, we need to adjust the argument **mean.model** of the specification. To improve readability, we split the model specifications into separate variables, as they are just lists. To include an ARCH term in the mean equation, the argument **archm** has to be set to **TRUE**. Further, as we could include the residual with different exponents, by setting **archpow = 2** we can be assured that only the squared residuals are included. The variance equation is set on the standard GARCH(1,1) model and then a specification object is created and estimation is executed as in the sections above.

[30] Note that due to the definition of the dummy variable as 0 for negative and 1 for positive values, the results can look different for other software solutions that define the dummy variable using 1 for negative values and 0 otherwise.

```
meanmodel = list(armaOrder=c(0,0),archm=T,archpow=2)
varmodel = list(garchOrder=c(1,1),model="sGARCH")
spec = ugarchspec(mean.model = meanmodel,variance.model = varmodel)
ugarchfit(spec,data=currencies$rjpy)
```

Running this code will produce the following output.

```
> meanmodel = list(armaOrder=c(0,0),archm=T,archpow=2)
> varmodel = list(garchOrder=c(1,1),model="sGARCH")
> spec = ugarchspec(mean.model = meanmodel,variance.model = varmodel)
> ugarchfit(spec,data=currencies$rjpy)

*---------------------------------*
*          GARCH Model Fit        *
*---------------------------------*

Conditional Variance Dynamics
-----------------------------------
GARCH Model     : sGARCH(1,1)
Mean Model      : ARFIMA(0,0,0)
Distribution    : norm

Optimal Parameters
------------------------------------
        Estimate  Std. Error   t value Pr(>|t|)
mu      0.011268    0.010090   1.11676  0.26410
archm  -0.027701    0.050193  -0.55188  0.58103
omega   0.001626    0.000156  10.44457  0.00000
alpha1  0.036768    0.000379  96.99471  0.00000
beta1   0.956564    0.001895 504.80490  0.00000

Robust Standard Errors:
        Estimate  Std. Error  t value Pr(>|t|)
mu      0.011268    0.012151   0.9273 0.353772
archm  -0.027701    0.053757  -0.5153 0.606345
omega   0.001626    0.000600   2.7096 0.006735
alpha1  0.036768    0.003722   9.8780 0.000000
beta1   0.956564    0.001787 535.3500 0.000000

LogLikelihood : -4344.631

Information Criteria
------------------------------------

Akaike       1.2182
Bayes        1.2230
Shibata      1.2182
Hannan-Quinn 1.2199
```

In this case, the estimated volatility parameter in the mean equation (**archm**) has a negative sign but is not statistically significant. We would thus conclude that, for these currency returns, there is no feedback from the conditional variance to the conditional mean.

18.4 Forecasting from GARCH Models

Reading: Brooks (2019, Section 9.18)

GARCH-type models can be used to forecast volatility. In this subsection, we will focus on generating the conditional variance forecasts in R. Let us assume that we want to generate forecasts based on the EGARCH model estimated earlier for the forecast period from "2016-08-03" to "2018-07-03". The first step is to re-estimate the EGARCH model for the subsample by adding the argument **out.sample=700** when calling the **ugarchfit** function. Doing this, the last 700 observations are not used for estimation.

```
spec = ugarchspec(mean.model = list(armaOrder=c(0,0)),variance.model = list(
   garchOrder=c(1,1),model="eGARCH"))
fit = ugarchfit(spec,data = currencies$rjpy,out.sample = 700)
```

Next, we generate the conditional variance forecasts. This can be done with the function **uGARCHforecast**. To compare static and dynamic forecasting methods, we will create two objects. On the one hand, a static forecast updates the current value and basically does 700 one-step ahead forecasts. On the other hand, a dynamic forecast uses the estimated one-step ahead forecast to derive the next step and creates the complete forecast without the out-of-sample observations. In R, and especially **uGARCHforecast** the difference between the two forecasting methods can be implemented by setting the arguments **n.ahead** and **n.roll**. For the static forecast, we set **n.ahead=1** to create one-step ahead forecasts. By also setting **n.roll=699**, the function rolls the in-sample period forward 699 times, such that we obtain 700 one-step ahead forecasts. For the dynamic forecasts, it suffices to set **n.ahead** to 700 and all of the forecasts are created. The following lines will achieve this.

```
static_fc = ugarchforecast(fit,n.ahead=1,n.roll = 699)
dynamic_fc = ugarchforecast(fit,n.ahead = 700)
```

Finally we want to graphically examine the conditional variance forecasts. As for the function created forecasts of the actual series, we can actually also look at the return forecasts. Both forecast series are saved in the slot **forecast**. After accessing it using , we can choose the variance forecast **sigmaFor** or the return forecast **seriesFor**. To plot both nicely in one graph, follow the code below.

```
par(lwd=2,cex.axis = 2)
x_axis = currencies$Date[currencies$Date >= "2016-08-03"]
plot(x_axis,static_fc@forecast$sigmaFor,type="l",xlab="",ylab="",col="blue3"
   )
lines(x_axis,dynamic_fc@forecast$sigmaFor,col="brown1")
legend("bottomleft", legend=c("Static", "Dynamic"),col=c("blue3","brown1"),
   lty= 1)
```

In the *Plots* window, the graph from Figure 22 is displayed.

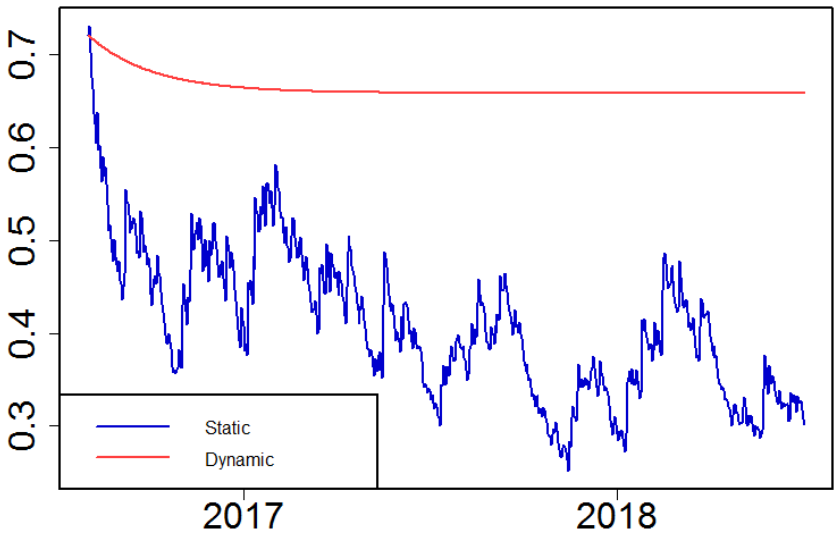


Figure 22: Graph of the Static and Dynamic Forecasts of the Conditional Variance

What do we observe? For the dynamic forecasts (red line), the value of the conditional variance starts from a historically high level at the end of the estimation period, relative to its unconditional average. Therefore, the forecasts converge upon their long-term mean value from above as the forecast horizon increases. Turning to the static forecasts (blue line), it is evident that the forecast period is characterised by a relatively low variance compared to the spikes seen at the end of the estimation period.

Note that while the forecasts are updated daily based on new information that feeds into the forecasts, the parameter estimates themselves are not updated. Thus, towards the end of the sample, the forecasts are based on estimates almost two years old. If we wanted to update the model estimates as we rolled through the sample, we would need to write some code to do this within a loop – it would also run much more slowly as we would be estimating a lot of GARCH models rather than one. Predictions can be similarly produced for any member of the GARCH family.

18.5 Estimation of Multivariate GARCH Models

Reading: Brooks (2019, Sections 9.20 and 9.21)

Multivariate GARCH models are in spirit very similar to their univariate counterparts, except that the former also specify equations for how the covariances move over time and are therefore by their nature inherently more complex to specify and estimate. To estimate a multivariate GARCH model in R, install the package **rmgarch**, which works in a similar fashion to **rugarch**. It allows us to estimate, simulate and forecast constant conditional correlation, dynamic conditional correlation, generalised orthogonal and Copula GARCH models. [31]

The first step for estimation is, as for univariate GARCH models, to specify the model in a specification object. For this purpose, we use the **rugarch** function **multispec**, so remember to load both packages into memory. With **multispec**, we can create a **uGARCHmultispec** object. This alone

[31] There is also the package **ccgarch**, which is independent of the two other packages and hence works in a different way. It extends to the family of CC-GARCH models.

could also be estimated using the univariate functions, and would result in a joint but separate estimation, hence the correlations between the currency rates would not be taken into account. To include an estimation of correlation, we further need to specify a **cGARCHspec** object, which can then be estimated by functions of the **rmgarch** package.

```
uspec = ugarchspec(mean.model = list(armaOrder=c(0,0)),variance.model = list
   (garchOrder=c(1,1),model="sGARCH"))
mspec = multispec(replicate(3,uspec))
cccspec = cgarchspec(mspec,VAR = F)
```

The steps described are implemented by running the three lines of code above. First, a univariate GARCH(1,1) is specified. Second, the univariate model is replicated three times to span a multiple GARCH(1,1) model. Third, the multiple GARCH model is extended to a multivariate GARCH model accounting for the correlations between variables. Note that no lags or exogenous variables are included, since the argument **VAR** is set to false. Hence, the mean equation will again just include an intercept term.

The estimation is done by the function **cgarchfit** which technically is used for Copula GARCH models, but without further specifications it reduces to a constant conditional correlation (CCC) model. Hence the command

```
mod = cgarchfit(cccspec,data = currencies[c("reur","rgbp","rjpy")])
mod
mod@mfit$Rt
```

is sufficient to estimate a CCC GARCH model for the three currency rates. The output should appear as on the next page after executing the above code.

First, a short summary of the estimated model is printed,, then the estimated parameters for the three variables are presented. Remember even though it says Copula GARCH Fit, the result is a static GARCH copula (Normal) model, and hence a constant conditional correlation GARCH model. As in the univariate case above, **mu** and **omega** denote the intercepts of the mean and variance equations, while **alpha1** and **beta1** are the coefficients of the lagged residual and variance, respectively. Lastly, the information criteria and the elapsed time are displayed.

```
> mod = cgarchfit(cccspec,data = currencies[c("reur","rgbp","rjpy")])
> mod

*---------------------------------------------------*
*                  Copula GARCH Fit                 *
*---------------------------------------------------*

Distribution            :  mvnorm
No. of Parameters       :  12
[VAR GARCH CC]          : [0+12+0]
No. of Series           :  3
No. of Observations     :  7141
Log-Likelihood          :  -9464.444
Av.Log-Likelihood       :  -1.325

Optimal Parameters
---------------------------------------------------
              Estimate  Std. Error     t value Pr(>|t|)
[reur].mu     -0.003892    0.004654    -0.83630 0.402987
[reur].omega   0.000272    0.000124     2.19080 0.028466
[reur].alpha1  0.020849    0.000967    21.56759 0.000000
[reur].beta1   0.978108    0.000040 24504.49445 0.000000
[rgbp].mu     -0.001328    0.004365    -0.30418 0.760991
[rgbp].omega   0.000827    0.000275     3.00635 0.002644
[rgbp].alpha1  0.034702    0.003231    10.73917 0.000000
[rgbp].beta1   0.961216    0.000506  1899.29421 0.000000
[rjpy].mu      0.006403    0.005269     1.21519 0.224295
[rjpy].omega   0.001640    0.000592     2.76890 0.005625
[rjpy].alpha1  0.036849    0.003524    10.45585 0.000000
[rjpy].beta1   0.956418    0.000614  1558.91794 0.000000

Information Criteria
--------------------

Akaike       2.6541
Bayes        2.6656
Shibata      2.6541
Hannan-Quinn 2.6581


Elapsed time : 3.411121

> mod@mfit$Rt
          [,1]      [,2]      [,3]
[1,] 1.0000000 0.6374866 0.3153877
[2,] 0.6374866 1.0000000 0.2195692
[3,] 0.3153877 0.2195692 1.0000000
>
```

The table is separated into different parts, organised by dependent variable. The header provides details on the estimation sample and reports a Wald test of the null hypothesis that all the coefficients on the independent variables in the mean equations are zero, which in our case is only the constant. The null hypothesis is not rejected even at the 10% level for any of the series. For each dependent variable, we first find estimates for the conditional mean equation, followed by the conditional variance estimates in a separate panel. It is evident that the parameter estimates in the variance equations are all both plausible and statistically significant. In the final panel, results for the conditional correlation parameters are reported. For example, the conditional correlation between the standardised residuals for 'reur' and 'rgbp' is estimated to be 0.637.

19 Modelling Seasonality in Financial Data

19.1 Dummy Variables for Seasonality

Reading: Brooks (2019, Section 10.3)

In this subsection, we will test for the existence of a 'January effect' in the stock returns of Microsoft using the '**macro.RData**' workfile. In order to examine whether there is indeed a January effect in a monthly time series regression, a dummy variable is created that takes the value one only in the months of January. To create the dummy **JANDUM**, it is easiest to use the package **lubridate** and its function **month** which extracts the month of a date variable. Then, we can set the value of **JANDUM** to the integer representing the binary value of the month being January or not. Within R, this can be coded as **macro$JANDUM = as.integer(month(macro$Date) == 1)**. The new variable 'JANDUM' contains the binary value of the logical expression 'month(macro$Date) == 1', which is false (0) for all months except January for which it is true and hence 1. We can now run the APT-style regression first used in section 7 but this time including the new 'JANDUM' dummy variable. The command for this regression is as follows:

```
summary(lm(ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney +
   dspread + rterm + APR00DUM + DEC00DUM +JANDUM, data = macro))
```

The results of this regression are presented below.

```
> summary(lm(ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney + dspread +
+              rterm + APR00DUM + DEC00DUM +JANDUM, data = macro))

Call:
lm(formula = ermsoft ~ ersandp + dprod + dcredit + dinflation +
    dmoney + dspread + rterm + APR00DUM + DEC00DUM + JANDUM,
    data = macro)

Residuals:
     Min       1Q   Median       3Q      Max
-20.4552  -4.4790  -0.2029   4.2176  24.1128

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.044217   0.494169   2.113 0.035260 *
ersandp       1.246083   0.090002  13.845  < 2e-16 ***
dprod        -0.284948   0.702759  -0.405 0.685365
dcredit      -0.010314   0.026110  -0.395 0.693045
dinflation    0.230073   1.368592   0.168 0.866589
dmoney        0.002951   0.015501   0.190 0.849110
dspread       1.188033   3.957871   0.300 0.764215
rterm         4.209344   1.635079   2.574 0.010427 *
APR00DUM    -37.797619   7.560603  -4.999 8.88e-07 ***
DEC00DUM    -28.862340   7.520532  -3.838 0.000146 ***
JANDUM        3.139099   1.654212   1.898 0.058518 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.466 on 372 degrees of freedom
  (2 observations deleted due to missingness)
Multiple R-squared:  0.4117,    Adjusted R-squared:  0.3959
F-statistic: 26.04 on 10 and 372 DF,  p-value: < 2.2e-16

>
```

As can be seen, the coefficient on the January dummy is statistically significant at the 10% level, and it has the expected positive sign. The coefficient value of 3.139, suggests that on average and holding everything else equal, Microsoft stock returns are around 3.1% higher in January than the average for other months of the year.

19.2 Estimating Markov Switching Models

Reading: Brooks (2019, Sections 10.5–10.7)

In this subsection, we will be estimating a Markov switching model in R. The example that we will consider relates to the changes in house prices series used previously. So we **load ukhp.RData**. Install and load the package **MSwM** for Markov switching models in R. We will use the function **msmFit** to model a Markov Switching model. However, for simplicity we do not include any exogenous variables, such that we only allow for the intercept and variance to vary between the two states.

To allow for these features we need to specify the input for the function **msmFit** in the following way.

```
msmodel = msmFit(lm(UKHP$dhp~1),k=2,sw=c(T,T))
```

The first argument is a linear model, in this case we regress the house price returns on only a constant. After that, the number of regimes **k=2** is defined. With the argument **sw** you can decide which parameters are allowed to vary between the regimes. The input is a binary vector which starts with the variable in the order of the regression and adds one more entry for the variance parameters. Since we also want the variance parameters to vary across states, we specify **sw** as a binary vector of two times **TRUE**.

After estimating the model, we print the results using **summary**. The output on the next page will appear.

```
> summary(msmodel)
Markov Switching Model

Call: msmFit(object = lm(UKHP$dhp ~ 1), k = 2, sw = c(T, T))

       AIC      BIC    logLik
  942.2982 961.4458 -469.1491

Coefficients:

Regime 1
---------
               Estimate Std. Error t value  Pr(>|t|)
(Intercept)(S)   0.7651     0.0698  10.961 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.8770494
Multiple R-squared:     0

Standardized Residuals:
        Min          Q1         Med          Q3         Max
-1.72813581 -0.40942855 -0.06541431  0.19823800  2.89598232

Regime 2
---------
               Estimate Std. Error t value Pr(>|t|)
(Intercept)(S)  -0.2372     0.1350  -1.757  0.07892 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.200559
Multiple R-squared: 5.345e-34

Standardized Residuals:
        Min          Q1         Med          Q3         Max
-3.28749095 -0.00474795  0.05597876  0.16141269  3.80457276

Transition probabilities:
           Regime 1  Regime 2
Regime 1 0.98354933 0.0240296
Regime 2 0.01645067 0.9759704
>
```

Examining the results, it is clear that the model has successfully captured the features of the data. Two distinct regimes have been identified: regime 1 with a negative mean return (corresponding to a price fall of 0.24% per month) and a relatively high volatility, whereas regime 2 has a high average price increase of 0.77% per month and a much lower standard deviation. At the end of the output, we can also see the transition probabilities. It appears that the regimes are fairly stable, with probabilities of around 98% of remaining in a given regime next period.

Finally, we would like to predict the probabilities of being in one of the regimes. We have only two regimes, and thus the probability of being in regime 1 tells us the probability of being in regime 2 at a given point in time, since the two probabilities must sum to one. The state probabilities can be accessed in the slot **filtProb** of the slot **Fit** of our model object. To obtain the probabilities for regime 2, we choose the second column of this matrix. To plot the probabilities against the time-line, just type:

```
par(lwd=2,cex.axis = 2)
plot(UKHP$Month,msmodel@Fit@filtProb[,2],type="l",xlab="",ylab="")
```

and you should obtain the graph in Figure 23.

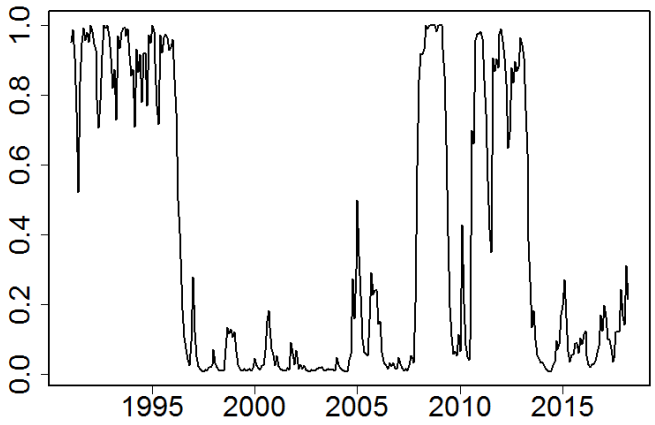


Figure 23: State Probabilities Graph

Examining how the graph moves over time, the probability of being in regime 2 was close to one until the mid-1990s, corresponding to a period of low or negative house price growth. The behaviour then changed and the probability of being in the low and negative growth state (regime 2) fell to zero and the housing market enjoyed a period of good performance until around 2005 when the regimes became less stable but tending increasingly towards regime 2. Since early 2013, it is again much more likely to be in regime 1. Note that the model object also contains a smoothed version of the state probabilities that can be accessed via **@Fit@smoProb**.

20 Panel Data Models

Reading: Brooks (2019, Chapter 11)

The estimation of panel models, with either fixed and random effects, is very easy with R; the harder part is organising the data so that the software can recognise that you have a panel of data and can apply the techniques accordingly. The method that will be adopted in this example, is to use the following three stages:

1. Set up your data in an Excel sheet so that it fits a panel setting, i.e., construct a variable that identifies the cross-sectional component (e.g., a company's CUSIP as identifier for different companies, a country code to distinguish between different countries, etc.), and a time variable, and stack the data for each company above each other. This is called the 'long' format.

2. Import the data into RStudio using any import function.

3. Declare the dataset to be panel data using the **plm** package.

The application to be considered here is that of a variant on an early test of the capital asset pricing model due to Fama and MacBeth (1973). Their test involves a 2-step estimation procedure: first, the betas are estimated in separate time-series regressions for each firm, and second, for each separate point in time, a cross-sectional regression of the excess returns on the betas is conducted

$$R_{it} - R_{ft} = \lambda_0 + \lambda_m \beta_{Pi} + u_i \tag{14}$$

where the dependent variable, $R_{it} - R_{ft}$, is the excess return of the stock i at time t and the independent variable is the estimated beta for the portfolio (P) that the stock has been allocated to. The betas of the firms themselves are not used on the RHS, but rather, the betas of portfolios formed on the basis of firm size. If the CAPM holds, then λ_0 should not be significantly different from zero and λ_m should approximate the (time average) equity market risk premium, $R_m - R_f$. Fama and MacBeth (1973) proposed estimating this second stage (cross-sectional) regression separately for each time period, and then taking the average of the parameter estimates to conduct hypothesis tests. However, one could also achieve a similar objective using a panel approach. We will use an example in the spirit of Fama–MacBeth comprising the annual returns and 'second pass betas' for 11 years on 2,500 UK firms.[32]

To test this model, we will use the '**panelx.xls**' workfile. Let us first have a look at the data in Excel. We see that missing values for the 'beta' and 'return' series are indicated by a '**NA**'. To account for this using the **read_excel** function, set **na = 'NA'**. Note that even forgetting about this, R will treat the values as 'NA', but it will produce a warning for every 'NA' value that was expected to be a numeric.

```
panelx <- read_excel("D:/Programming Guide/data/panelx.xls",col_types = c("
   numeric","numeric","numeric","numeric"),na = 'NA')
library(plm)
data = pdata.frame(panelx, index=c("firm_ident", "year"))
pdim(data)
```

[32] Source: computation by Keith Anderson and the author. There would be some significant limitations of this analysis if it purported to be a piece of original research, but the range of freely available panel datasets is severely limited and so hopefully it will suffice as an example of how to estimate panel models with R. No doubt readers, with access to a wider range of data, will be able to think of much better applications.

After importing the data set, we transform it to a panel data set using the function **pdata.frame** which only needs the imported set **panelx** and the argument **index** to identify firm-year observations. With the command **pdim(data)** we can check whether the panel is balanced and it will answer the following.

```
> pdim(data)
Balanced Panel: n = 2500, T = 11, N = 27500
>
```

Now our dataset is ready to be used for panel data analysis. Let us have a look at the summary statistics for the two main variables by applying the **summary** function.

```
> summary(data[c("return","beta")])
     return              beta
 Min.   :-1.005   Min.   :0.661
 1st Qu.:-0.005   1st Qu.:0.941
 Median :-0.004   Median :1.082
 Mean   :-0.002   Mean   :1.105
 3rd Qu.:-0.003   3rd Qu.:1.249
 Max.   : 0.706   Max.   :1.612
 NA's   :3409     NA's   :18427
>
```

This is always a good way to check whether data was imported correctly. However, our primary aim is to estimate the CAPM-style model in a panel setting. Let us first estimate a simple pooled regression with neither fixed nor random effects. Note that in this specification we are basically ignoring the panel structure of our data and assuming that there is no dependence across observations (which is very unlikely for a panel dataset). We can use the standard **lm** function to estimate the model, but let us introduce the **plm** function from the **plm** package. It works in similar fashion to **lm**, but needs a panel data set as input, which is why we created the panel data set **data**. Specifying the formula to regress returns on betas is the first input. After that, we need to tell **plm** what kind of regression is to be estimated since the function can also estimate random and fixed effects. For a pooled regression, set **model="pooling"**. Finally, we put the code together as follows

```
pooled = plm(return~beta, model="pooling", data=data)
summary(pooled)
```

and after running the two lines above, we should find the output on the next page.

We can see that neither the intercept nor the slope is statistically significant. The returns in this regression are in proportion terms rather than percentages, so the slope estimate of 0.000454 corresponds to a risk premium of 0.0454% per month, or around 0.5% per year, whereas the average excess return across all firms in the sample is around 2.2% per year.

```
> pooled = plm(return~beta, model="pooling", data=data)
> summary(pooled)
Pooling Model

Call:
plm(formula = return ~ beta, data = data, model = "pooling")

Unbalanced Panel: n = 1734, T = 1-11, N = 8856

Residuals:
      Min.    1st Qu.     Median    3rd Qu.       Max.
-1.0073394 -0.0133705 -0.0017671  0.0201393  0.7041667

Coefficients:
             Estimate Std. Error t-value Pr(>|t|)
(Intercept) 0.00184254 0.00307461  0.5993    0.549
beta        0.00045439 0.00273471  0.1662    0.868

Total Sum of Squares:    24.205
Residual Sum of Squares: 24.204
R-Squared:      3.1181e-06
Adj. R-Squared: -0.00010982
F-statistic: 0.0276078 on 1 and 8854 DF, p-value: 0.86804
>
```

But this pooled regression assumes that the intercepts are the same for each firm and for each year. This may be an inappropriate assumption. Thus, next we (separately) introduce fixed and random effects to the model. The only aspect to change is the argument **model** to **"within"** in the above specification.

```
> fixed = plm(return~beta, model="within", data=data)
> summary(fixed)
Oneway (individual) effect Within Model

Call:
plm(formula = return ~ beta, data = data, model = "within")

Unbalanced Panel: n = 1734, T = 1-11, N = 8856

Residuals:
     Min.   1st Qu.    Median   3rd Qu.      Max.
-0.891228 -0.015834  0.000000  0.016408  0.653811

Coefficients:
       Estimate Std. Error t-value Pr(>|t|)
beta -0.0118931  0.0041139 -2.8909 0.003852 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Total Sum of Squares:    18.371
Residual Sum of Squares: 18.35
R-Squared:      0.0011723
Adj. R-Squared: -0.24205
F-statistic: 8.35756 on 1 and 7121 DF, p-value: 0.0038525
>
```

We can see that the estimate on the beta parameter is negative and statistically significant here. An intercept is not reported as there are 1734 groups (firms), each with different intercepts (fixed effects).

We now estimate a random effects model. For this, we simply change the argument **model** to **"random"** in the **plm** function. We leave all other specifications unchanged and press **Enter** to

generate the regression output.

The slope estimate is again of a different order of magnitude compared to both the pooled and the fixed effects regressions. As the results for the fixed effects and random effects models are quite different, it is of interest to determine which model is more suitable for our setting. To check this, we use the Hausman test. The null hypothesis of the Hausman test is that the random effects (RE) estimator is indeed an efficient (and consistent) estimator of the true parameters. If this is the case, there should be no systematic difference between the RE and FE estimators and the RE estimator would be preferred as the more efficient technique. In contrast, if the null is rejected, the fixed effect estimator needs to be applied.

```
> random = plm(return~beta, model="random", data=data)
> summary(random)
Oneway (individual) effect Random Effect Model
   (Swamy-Arora's transformation)

Call:
plm(formula = return ~ beta, data = data, model = "random")

Unbalanced Panel: n = 1734, T = 1-11, N = 8856

Effects:
                     var   std.dev share
idiosyncratic 0.0025768 0.0507625 0.944
individual    0.0001529 0.0123657 0.056
theta:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.02841 0.12183 0.19262 0.16455 0.22215 0.22215

Residuals:
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-0.98141 -0.01314 -0.00156  0.00031  0.01972  0.69480

Coefficients:
             Estimate Std. Error z-value Pr(>|z|)
(Intercept)  0.0032809  0.0032887  0.9976   0.3185
beta        -0.0014994  0.0029132 -0.5147   0.6068

Total Sum of Squares:    23.117
Residual Sum of Squares: 23.125
R-Squared:      0.00048503
Adj. R-Squared: 0.00037214
Chisq: -2.85702 on 1 DF, p-value: 1
>
```

To run the Hausman test, we use the function **phtest** which only needs the two models as inputs. As we have stored the two models before, it is sufficient to run the test by typing **phtest(fixed, random)**. The output with the Hausman test results should appear as below.

```
> phtest(fixed, random)

        Hausman Test

data:  return ~ beta
chisq = 12.804, df = 1, p-value = 0.0003459
alternative hypothesis: one model is inconsistent

>
```

The χ^2 value for the Hausman test is 12.804 with a corresponding p-value of 0.0003. Thus, the null hypothesis that the difference in the coefficients is not systematic is rejected at the 1% level, implying that the random effects model is not appropriate and that the fixed effects specification is to be preferred.

21 Limited Dependent Variable Models

Reading: Brooks (2019, Chapter 12)

Estimating limited dependent variable models in R is very simple. The example that will be considered here concerns whether it is possible to determine the factors that affect the likelihood that a student will fail his/her MSc. The data comprise a sample from the actual records of failure rates for five years of MSc students at the ICMA Centre, University of Reading, contained in the spreadsheet 'MSc_fail.xls'. While the values in the spreadsheet are all genuine, only a sample of 100 students is included for each of the five years who completed (or not as the case may be!) their degrees in the years 2003 to 2007 inclusive. Therefore, the data should not be used to infer actual failure rates on these programmes. The idea for this is taken from a study by Heslop and Varotto (2007), which seeks to propose an approach to prevent systematic biases in admissions decisions.[33]

The objective here is to analyse the factors that affect the probability of failure of the MSc. The dependent variable ('fail') is binary and takes the value 1 if that particular candidate failed at first attempt in terms of his/her overall grade and 0 elsewhere. Therefore, a model that is suitable for limited dependent variables is required, such as a logit or probit.

The other information in the spreadsheet that will be used includes the age of the student, a dummy variable taking the value 1 if the student is female, a dummy variable taking the value 1 if the student has work experience, a dummy variable taking the value 1 if the student's first language is English, a country code variable that takes values from 1 to 10,[34] a dummy that takes the value 1 if the student already has a postgraduate degree, a dummy variable that takes the value 1 if the student achieved an A-grade at the undergraduate level (i.e., a first-class honours degree or equivalent), and a dummy variable that takes the value 1 if the undergraduate grade was less than a B-grade (i.e., the student received the equivalent of a lower second-class degree). The B-grade (or upper second-class degree) is the omitted dummy variable and this will then become the reference point against which the other grades are compared.

The reason why these variables ought to be useful predictors of the probability of failure should be fairly obvious and is therefore not discussed. To allow for differences in examination rules and in average student quality across the five-year period, year dummies for 2004, 2005, 2006 and 2007 are created and thus the year 2003 dummy will be omitted from the regression model.

First, we import the dataset into R. All variables should be in the numeric format and overall there should be 500 observations in the dataset for each series with no missing observations.

To begin with, suppose that we estimate a linear probability model of Fail on a constant, Age, English, Female, Work experience, A-Grade, Below-B-Grade, PG-Grade and the year dummies. Running the **lm** method for this would yield the following results.

[33] Note that since this example only uses a subset of their sample and variables in the analysis, the results presented below may differ from theirs. Since the number of fails is relatively small, I deliberately retained as many fail observations in the sample as possible, which will bias the estimated failure rate upwards relative to the true rate.

[34] The exact identities of the countries involved are not revealed in order to avoid any embarrassment for students from countries with high relative failure rates, except that Country 8 is the UK!

```
> summary(lm(Fail ~ Age + English + Female + WorkExperience + Agrade + BelowBGrade + PGDegree +
Year2004 + Year2005 + Year2006 + Year2007, data = MSc_fail))

Call:
lm(formula = Fail ~ Age + English + Female + WorkExperience +
    Agrade + BelowBGrade + PGDegree + Year2004 + Year2005 + Year2006 +
    Year2007, data = MSc_fail)

Residuals:
     Min       1Q   Median       3Q      Max
-0.40904 -0.17112 -0.10499 -0.03341  0.98565

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)     0.103881   0.120528   0.862  0.38918
Age             0.001322   0.004336   0.305  0.76059
English        -0.020073   0.031528  -0.637  0.52463
Female         -0.029380   0.035053  -0.838  0.40235
WorkExperience -0.062028   0.031436  -1.973  0.04904 *
Agrade         -0.080700   0.037720  -2.139  0.03289 *
BelowBGrade     0.092616   0.050226   1.844  0.06579 .
PGDegree        0.028661   0.047410   0.605  0.54576
Year2004        0.056910   0.047751   1.192  0.23392
Year2005       -0.011101   0.048367  -0.230  0.81856
Year2006        0.141581   0.048034   2.948  0.00336 **
Year2007        0.085150   0.049727   1.712  0.08747 .
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3332 on 488 degrees of freedom
Multiple R-squared:  0.06625,   Adjusted R-squared:  0.0452
F-statistic: 3.148 on 11 and 488 DF,  p-value: 0.0003962

>
```

While this model has a number of very undesirable features as discussed in Brooks (2019), it would nonetheless provide a useful benchmark with which to compare the more appropriate models estimated below. Next, we estimate a logit model and a probit model using the same dependent and independent variables as above. Both models can be estimated using the generalised linear modelling function **glm**. We begin with the logit model by specifying the argument **family** in **glm** as **binmoial("logit")**, keeping everything else as before.

```
> logit = glm(formula(linear),data = MSc_fail, family = binomial("logit"))
> summary(logit)

Call:
glm(formula = formula(linear), family = binomial("logit"), data = MSc_fail)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.2275  -0.5870  -0.4228  -0.2980   2.5579

Coefficients:
               Estimate Std. Error z value Pr(>|z|)
(Intercept)    -2.25637    1.07300  -2.103  0.03548 *
Age             0.01101    0.03813   0.289  0.77275
English        -0.16512    0.28295  -0.584  0.55952
Female         -0.33389    0.34923  -0.956  0.33902
WorkExperience -0.56877    0.28847  -1.972  0.04865 *
Agrade         -1.08503    0.49110  -2.209  0.02715 *
BelowBGrade     0.56235    0.37351   1.506  0.13217
PGDegree        0.21208    0.41990   0.505  0.61350
Year2004        0.65321    0.50092   1.304  0.19223
Year2005       -0.18382    0.58794  -0.313  0.75454
Year2006        1.24658    0.47365   2.632  0.00849 **
Year2007        0.85042    0.49705   1.711  0.08710 .
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 393.92  on 499  degrees of freedom
Residual deviance: 359.43  on 488  degrees of freedom
AIC: 383.43

Number of Fisher Scoring iterations: 5

>
```

Next, we estimate the above specification as a probit model. We input the same model specifications as in the logit case and change the **family** argument to**binomial(“probit”)**.

```
> probit = glm(formula(linear),data = MSc_fail, family = binomial("probit"))
> summary(probit)

Call:
glm(formula = formula(linear), family = binomial("probit"), data = MSc_fail)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.2066  -0.5872  -0.4269  -0.2792   2.6302

Coefficients:
                Estimate Std. Error z value Pr(>|z|)
(Intercept)    -1.287212   0.583782  -2.205  0.02746 *
Age             0.005677   0.020910   0.272  0.78601
English        -0.093792   0.154685  -0.606  0.54429
Female         -0.194107   0.185278  -1.048  0.29480
WorkExperience -0.318247   0.156118  -2.039  0.04150 *
Agrade         -0.538813   0.235001  -2.293  0.02186 *
BelowBGrade     0.341802   0.215586   1.585  0.11286
PGDegree        0.132957   0.229911   0.578  0.56306
Year2004        0.349664   0.255411   1.369  0.17099
Year2005       -0.108329   0.289379  -0.374  0.70814
Year2006        0.673613   0.245478   2.744  0.00607 **
Year2007        0.433786   0.257441   1.685  0.09199 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 393.92  on 499  degrees of freedom
Residual deviance: 358.91  on 488  degrees of freedom
AIC: 382.91

Number of Fisher Scoring iterations: 6

>
```

Turning to the parameter estimates on the explanatory variables, we find that only the work experience and A-grade variables and two of the year dummies have parameters that are statistically significant, and the Below B-grade dummy is almost significant at the 10% level in the probit specification (although less so in the logit model). However, the proportion of fails in this sample is quite small (13.4%), which makes it harder to fit a good model than if the proportion of passes and fails had been more evenly balanced.

A test on model adequacy is to produce a set of in-sample forecasts – in other words, to look at the fitted values. To visually inspect the fitted values, we plot them as they are a variable of the model object. The following two lines of code are sufficient to do so.

```
par(cex.axis=1.5)
plot(probit$fitted.values,type="l",xlab="",ylab="")
```

The resulting plot should resemble that in Figure 24.

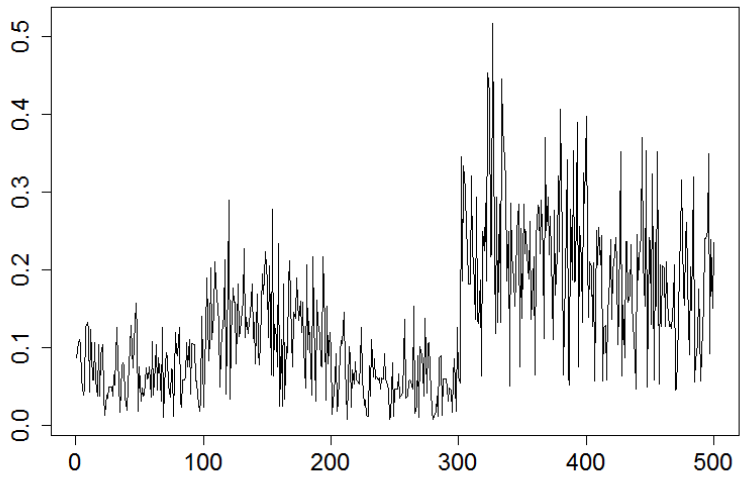


Figure 24: Graph of the Fitted Values from the Failure Probit Regression

It is important to note that we cannot interpret the parameter estimates in the usual way (see discussion in Brooks (2019)). In order to be able to do this, we need to calculate the marginal effects. For this purpose, the **margins** package is very helpful. After installing and including it, we simply apply the function **margins** on the two model objects and R should generate the table of marginal effects and corresponding statistics as shown below.

```
> margins(logit)
Average marginal effects
glm(formula = formula(linear), family = binomial("logit"), data = MSc_fail)

      Age  English   Female WorkExperience  Agrade BelowBGrade PGDegree Year2004 Year2005
 0.001186 -0.01779 -0.03597       -0.06127 -0.1169     0.06058  0.02285  0.07036  -0.0198
 Year2006 Year2007
   0.1343  0.09161
> margins(probit)
Average marginal effects
glm(formula = formula(linear), family = binomial("probit"), data = MSc_fail)

      Age  English   Female WorkExperience  Agrade BelowBGrade PGDegree Year2004 Year2005
 0.001118 -0.01847 -0.03822       -0.06267 -0.1061      0.0673  0.02618  0.06885 -0.02133
 Year2006 Year2007
   0.1326  0.08542
>
```

Looking at the results, we find that not only are the marginal effects for the probit and logit models quite similar in value, they also closely resemble the coefficient estimates obtained from the linear probability model estimated earlier in the section.

Now that we have calculated the marginal effects, these values can be intuitively interpreted in terms of how the variables affect the probability of failure. For example, an age parameter value of around 0.0012 implies that an increase in the age of the student by one year would increase the probability of failure by 0.12%, holding everything else equal, while a female student is around 3.5% less likely than a male student with otherwise identical characteristics to fail. Having an A-grade (first class) in the bachelor's degree makes a candidate around 11% less likely to fail than an otherwise identical student with a B-grade (upper second-class degree). Finally since the year 2003 dummy has been omitted from

the equations, this becomes the reference point. So students were more likely in 2004, 2006 and 2007, but less likely in 2005, to fail the MSc than in 2003.

22 Simulation Methods

22.1 Deriving Critical Values for a Dickey–Fuller Test Using Simulation

Reading: Brooks (2019, Sections 13.1–13.7)

In this and the following subsections we will use simulation techniques in order to model the behaviour of financial series. In this first example, our aim is to develop a set of critical values for Dickey-Fuller test regressions. Under the null hypothesis of a unit root, the test statistic does not follow a standard distribution, and therefore a simulation would be required to obtain the relevant critical values. Obviously, these critical values are well documented, but it is of interest to see how one could generate them. A very similar approach could then potentially be adopted for situations where there has been less research and where the results are relatively less well known.

The simulation would be conducted in the following four steps:

1. Construct the data generating process under the null hypothesis – that is obtain a series for y that follows a unit root process. This would be done by:
   - Drawing a series of length T, the required number of observations, from a normal distribution. This will be the error series, so that $u_t \sim N(0, 1)$.
   - Assuming a first value for y, i.e., a value for y at time $t = 1$.
   - Constructing the series for y recursively, starting with y_2, y_3, and so on

$$\begin{aligned} y_2 &= y_1 + u_2 \\ y_3 &= y_2 + u_3 \\ &\vdots \\ y_T &= y_{T-1} + u_T \end{aligned}$$

2. Calculate the test statistic, τ.
3. Repeat steps 1 and 2 N times to obtain N replications of the experiment. A distribution of values for τ will be obtained across the replications.
4. Order the set of N values of τ from the lowest to the highest. The relevant 5% critical value will be the 5th percentile of this distribution.

Some R code for conducting such a simulation is given below. The simulation framework considers a sample of 1,000 observations and Dickey–Fuller regressions with no constant or trend, a constant but no trend, and a constant and a trend. 50,000 replications are used in each case, and the critical values for a one-sided test at the 1%, 5% and 10% levels are determined. The code can be found pre-written in R script entitled **'dickey_fuller_simulation.R'**.

The R script is a list of commands that could be typed into the *Console* in the same way, hence it is just a way of saving your code in a file. In RStudio, you can easily open the file by clicking **File/Open File...** in the main menu. To run the script, you can either press the **Run** button in the top line in RStudio or use the Main menu option **Code/** and then choose to run selected lines, the whole code, or even sections of it. The quickest way however, is to mark the code and press **Ctrl + Enter**. In this way, you can also go through the code line by line, which is sometimes helpful for debugging.

The appearance of the code in the *Source* window and *Console* can be changed within RStudio by clicking **Tool/Global Options...**. In the pop up window, choose the tab **Appearance** as shown in

Figure 25 and you will find several editor themes. While the default scheme does only use blue and green colours to highlight different parts of code, the reader might find it easier to read code using a different style. However, within this guide the default option will be kept.

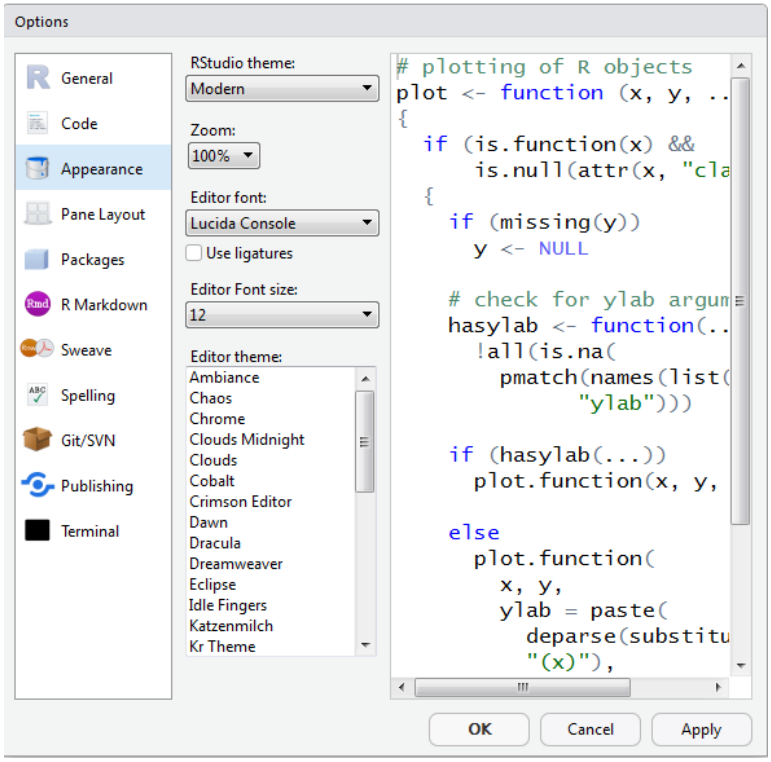


Figure 25: Changing the Way Code is Displayed in RStudio

The different colours indicate different characteristics of the instructions. In the default theme, `blue` represents the main key words for loops (for, while), if-else statements and the definition of functions. Also, any numbers are displayed in `blue` as are key words such as (TRUE, FALSE, NULL or NA). Expressions in `green` mark comments preceded by a hash tag (#) as well as strings between double (" ") or single quote marks (' '). Comments are not part of the actual instructions but rather serve to explain and describe the code.

The lines of code on the next page are taken from the R script **'dickey_fuller_simulation.R'** which creates critical values for the Dickey–Fuller test. The discussion below explains the functions of each command line.

The first line is a simple comment that explains the purpose and contents of the do-file.[35] The lines that follow contain the actual commands that perform the manipulations of the data. The first couple of lines are mere preparation for the main simulation but are a necessary to access the simulated critical values later on.

Line 2, **'set.seed(123456)'**, sets the so-called random number seed. This is necessary to be able to replicate the exact t-values created with this program on any other computer at a later time and the command serves to define the starting value for draws from a standard normal distribution which are necessary in later stages to create variables that follow a standard normal distribution. Line 3 creates the three variables in which we will store the t-statistics and sets them to **NULL**. Note that the names already reveal which model is used. It is very helpful to use explanatory variable names instead of generic names such as **t1**, to improve the readability of the code.

[35] Adding comments to your do-files is a useful practice and proves to be particularly useful if others will use your code, or if you revisit analyses that you have first carried out some time ago, as they help you to recall the logic of the commands and steps.

```
1  # Script to generate critical values for Dickey Fuller test
2  set.seed(123456)
3  tnone = tconst = ttrend = NULL
4
5  for (i in 1:50000){
6    y = c(0, cumsum(rnorm(1199)))
7    dy = c(NA,diff(y))
8    lagy = c(NA,y[1:1199])
9    # Model with no intercept
10   lmnone = lm(dy[-(1:200)] ~ 0 + lagy[-(1:200)])
11   tnone = c(tnone, coef(summary(lmnone))[1]/coef(summary(lmnone))[2])
12   # Model with intercept, no trend
13   lmconst = lm(dy[-(1:200)] ~ 1 + lagy[-(1:200)])
14   tconst = c(tconst,coef(summary(lmconst))[2,1]/coef(summary(lmconst))[2,2])
15   # Model with intercept and trend
16   lmtrend = lm(dy[-(1:200)] ~ 1 + lagy[-(1:200)] + c(1:1000))
17   ttrend = c(ttrend,coef(summary(lmtrend))[2,1]/coef(summary(lmtrend))[2,2])
18 }
19 quantile(tnone,c(0.01,0.05,0.1))
20 quantile(tconst,c(0.01,0.05,0.1))
21 quantile(ttrend,c(0.01,0.05,0.1))
```

Line 5 sets up the conditions for the loop, i.e., the number of repetitions N that will be performed. Loops are always indicated by braces; the set of commands over which the loop is performed is contained within the curly brackets (**{ }**). For example, in our command the loop ends in line 18.

Before turning to the specific conditions of the loop, let us have a look at the set of commands that we want to perform the loop over, i.e., the commands that generate the t-values for the Dickey–Fuller regressions. They are stated in lines 6 to 17. Line 6 creates a random series of 1199 numbers with a standard normal distribution **rnorm(1199)** of which we write the cumulated sums into a vector **y** starting with 0. Hence, we end up with a vector comprising 1200 entries that resembles a random walk-series that follows a unit root process. Recall that a random walk process is defined as the past value of the variable plus a standard normal error term. The next lines, 7 and 8, write the first differences and lagged values of **y** into the variables **dy** and **lagy**, adding an 'NA' value at the start to keep the same length.

Lines 9 to 16 are commands to run linear regressions and compute the t-statistics. They are separated into three blocks for each model consisting of two lines. In the first line, respectively, 10, 13 or 16, the linear model is estimated. The second line, respectively, 11, 14 or 17, computes the t-statistic and appends it to the corresponding vector.

Note that when generating random draws, it sometimes takes a while before the constructed series have the properties that they should, and therefore, when generating the regressions, we exclude the first 200 observations. Therefore the dependent variable is always **dy[-(1:200)]**, so the series comprises the first differences without the first 200 observations. The independent variables change for the different models. In line 10, only the lagged values of y, hence the values starting at 200, are used. To avoid an intercept, we need to specify the model as **dy[-(1:200)]** $\sim$ **0 + lagy[-(1:200)]**. In line 13, we change this to include an intercept and write 1 instead of 0. Note that this is only to emphasise the difference between the lines, since the intercept is included by default anyway. Lastly, in line 16, the model includes an intercept and a time trend, which is incorporated by adding the simple vector **c(1:1000)**.

Lines 11, 14 and 17 generate the t-values corresponding to the particular models. It is the formula

for computing the t-value, namely the coefficient estimate on **'lagy'** divided by the standard error of the coefficient. We can extract these coefficients using the function **coef** applied to the summary of the linear model. Note that for the multiple variable models **lmconst** and **lmtrend**, you need to also specify the column, since the output is a coefficient matrix, while for **lmnone** there is only one coefficient.

If we were to execute this set of commands one time, we would generate one t-value for each of the models. However, our aim is to get a large number of t-statistics in order to have a distribution of values. Thus, we need to repeat the set of commands for the desired number of repetitions. This is done by the loop command **for** in line 5. It states that the set of commands included in the braces will be executed 50,000 times (**'i in 1:50000'**). Note that, for each of these 50,000 repetitions, a new set of t-values will be generated and added to the vectors **tnone, tconst** and **ttrend**.

Finally, after the last repetition of the commands loop has been executed, in lines 19 to 21, we compute the 1st, 5th and 10th percentile for the three variables 'tnone', 'tconst', 'ttrend' using the **quantile** function.

The critical values obtained by running the above program, which are virtually identical to those found in the statistical tables in Brooks (2019), are presented in the output below. This is to be expected, for the use of 50,000 replications should ensure that an approximation to the asymptotic behaviour is obtained. For example, the 5% critical value for a test regression with no constant or trend and 500 observations is −1.93 in this simulation, and −1.95 in Fuller (1976).

```
> quantile(tnone,c(0.01,0.05,0.1))
       1%        5%       10%
-2.586771 -1.944655 -1.621259
> quantile(tconst,c(0.01,0.05,0.1))
       1%        5%       10%
-3.430630 -2.859899 -2.565954
> quantile(ttrend,c(0.01,0.05,0.1))
       1%        5%       10%
-3.968928 -3.414297 -3.130338
>
```

Although the Dickey–Fuller simulation was unnecessary in the sense that the critical values for the resulting test statistics are already well known and documented, a very similar procedure could be adopted for a variety of problems. For example, a similar approach could be used for constructing critical values or for evaluating the performance of statistical tests in various situations.

22.2 Pricing Asian Options

Reading: Brooks (2019, Section 13.8)

In this subsection, we will apply Monte Carlo simulations to price Asian options. The steps involved are:

1. *Specify a data generating process for the underlying asset.* A random walk with drift model is usually assumed. Specify also the assumed size of the drift component and the assumed size of the volatility parameter. Specify also a strike price K, and a time to maturity, T.

2. Draw a series of length T, the required number of observations for the life of the option, from a normal distribution. This will be the *error series*, so that $\epsilon_t \sim N(0, 1)$.

3. Form a series of observations of length T on the *underlying asset*.

4. *Observe the price of the underlying asset at maturity observation* T. For a call option, if the value of the underlying asset on maturity date $P_t \leq K$, the option expires worthless for this replication. If the value of the underlying asset on maturity date $P_t > K$, the option expires in the money, and has a value on that date equal to $P_T - K$, which should be discounted back to the present using the risk-free rate.

5. Repeat steps 1 to 4 a total of N times, and take the average value of the option over N replications. This average will be the *price of the option*.

A sample of R code for determining the value of an Asian option is given below. The example is in the context of an arithmetic Asian option on the FTSE 100, and two simulations will be undertaken with different strike prices (one that is out of the money forward and one that is in the money forward). In each case, the life of the option is six months, with daily averaging commencing immediately, and the option value is given for both calls and puts in terms of index options. The parameters are given as follows, with dividend yield and risk-free rates expressed as percentages:

Simulation 1:	Strike = 6500	**Simulation 2:**	Strike = 5500
	Spot FTSE = 6,289.70		Spot FTSE = 6,289.70
	Risk-free rate = 6.24		Risk-free rate = 6.24
	Dividend yield = 2.42		Dividend yield = 2.42
	Implied Volatility = 26.52		Implied Volatility = 34.33

All experiments are based on 25,000 replications and their antithetic variates (total: 50,000 sets of draws) to reduce Monte Carlo sampling error. Some sample code for pricing an Asian option for normally distributed errors is given on the next page.

Some parts of the program use identical instructions to those given for the Dickey–Fuller critical value simulation, and so annotation will now focus on the construction of the program and on previously unseen commands. As with the R script for the DF critical value simulation, you can open (and run) the program to price Asian options by clicking **File/Open File...**, choosing the R file **'asian_option_pricing.R'** then pressing the **Run** button in RStudio. You should then be able to inspect the set of commands and identify commands, comments and other operational variables based on the colouring system described in the previous subsection.

The first lines set the random seed number and the values for the simulation that were given, i.e., spot price at $t = 0$ (**s0**), the risk-free rate (**rf**), the dividend yield (**dy**) and the number of observations (**obs**). For a specific scenario, there is also the strike price **(K)** and the implied volatility (**iv**) given. Note that scenario 2 is in a comment line and would not be executed, as this code would run for scenario

1. To change this, just remove the hash tag in line 5 and add one at the beginning of line 4. It can be helpful to use the semicolon here, to write multiple commands in one line and as for line 6, this saves a bit of space to improve the overview of the program. However, avoid the use of a semicolon between two meaningful operations.

```
1  # Script for Monte-Carlo Simulation of Asian Option
2  set.seed(123456)
3  # Initial values and derived constants
4  K = 6500; iv = 0.2652 # Simulation one
5  # K = 5500; iv = 0.3433 # Simulation two
6  s0 = 6289.7; rf = 0.0624; dy = 0.0242; ttm = 0.5; obs = 125
7  dt = ttm/obs
8  drift = (rf-dy-iv^2/2)*dt
9  vsqrdt = iv*dt^0.5
10
11 putval = callval = NULL
12 for (i in 1:25000){
13   random = rnorm(obs) # create cumulative sum of random numbers
14   # Spot price evolution for positive and negative random numbers
15   spot = s0*exp(drift*c(1:obs)+vsqrdt*cumsum(random))
16   spot_neg = s0*exp(drift*c(1:obs)+vsqrdt*cumsum(-random))
17   # Compute call values
18   callval = c(callval, max(mean(spot)-K,0)*exp(-rf*ttm))
19   callval = c(callval, max(mean(spot_neg)-K,0)*exp(-rf*ttm))
20   # Compute Put values
21   putval = c(putval, max(K-mean(spot),0)*exp(-rf*ttm))
22   putval = c(putval, max(K-mean(spot_neg),0)*exp(-rf*ttm))
23 }
24 mean(callval)
25 mean(putval)
```

Line 12 indicates that we will be performing **N=25,000** repetitions of the set of commands (i.e., simulations). However, we will still generate 50,000 values for the put and call options each by using a 'trick', i.e., using the antithetic variates. This will be explained in more detail once we get to this command. Overall, each simulation will be performed for a set of 125 observations (see variable **obs** in line 6).

The model assumes, under a risk-neutral measure, that the underlying asset price follows a geometric Brownian motion, which is given by

$$dS = (\text{rf} - \text{dy})Sdt + \sigma dz, \tag{15}$$

where dz is the increment of a Brownian motion. The discrete time approximation to this for a time step of one can be written as

$$S_t = S_{t-1} \exp\left[\left(\text{rf} - \text{dy} - \frac{1}{2}\sigma^2\right)dt + \sigma\sqrt{dt}u_t\right] \tag{16}$$

where u_t is a white noise error process.

Lines 13 and 15 generate the path of the underlying asset. First, 125 random $\mathcal{N}(0,1)$ draws are made and written to the variable **random**. In line 15, the complete path for the next 125 observations is computed. The code makes use of ability of R to vectorise many functions, i.e., they can be applied to every entry of vector instead of running through a loop for every entry. This makes the code much more time efficient.

Instead of defining every one of the 125 steps recursively, we use the whole vector **random**. With the cumulative sum **cumsum**, we add up the exponents in **random** and multiplying **drift** with a vector from 1 to 125 adds a drift to the exponent for every step. The process then repeats using the antithetic variates, constructed using **-random**. The spot price series based on the antithetic variates is written to **spot_neg**. This way we can double the 'simulated' values for the put and call options without having to draw further random variates and run double the number of loop iterations.

Finally, lines 18 and 19 (21 and 22) compute the value of the call (put). For an Asian option, this equals the average underlying price less the strike price, if the average is greater than the strike (i.e., if the option expires in the money), and zero otherwise. Vice versa for the put. The payoff at expiry is discounted back to the present based on the risk-free rate (using the expression **exp(-rf*ttm)**). This is done for the paths derived from the positive and negative random draws and appended to the vectors **callval** and **putval**, respectively.

This completes one cycle of the loop, which will then be repeated for further 24,999 times and overall creates 50,000 values for each of the call and put options. Once the loop has finished, the option prices are calculated as the averages over the 50,000 replications in lines 24 and 25.
For the specifics stated above, the call price is approximately 204.68 and the put price lies around 349.62. Note that both call values and put values can be calculated easily from a given simulation, since the most computationally expensive step is in deriving the path of simulated prices for the underlying asset. In the following table, we compare the simulated call and put prices for different implied volatilities and strike prices along with the values derived from an analytical approximation to the option price, derived by Levy, and estimated using VBA code in Haug (1998), pp. 97–100.

Table 2: Simulated Asian Option Prices

Simulation 1: Strike = 6500, IV = 26.52%		**Simulation 2:** Strike = 5500, IV = 34.33%	
Call	Price	**Call**	Price
Analytical Approximation	203.45	Analytical Approximation	888.55
Monte Carlo Normal	204.68	Monte Carlo Normal	886.03
Put	Price	**Put**	Price
Analytical Approximation	348.70	Analytical Approximation	64.52
Monte Carlo Normal	349.62	Monte Carlo Normal	61.81

In both cases, the simulated option prices are quite close to the analytical approximations, although the Monte Carlo seems to overvalue the out-of-the-money call and to undervalue the out-of-the-money put. Some of the errors in the simulated prices relative to the analytical approximation may result from the use of a discrete-time averaging process using only 125 data points.

23 Value at Risk

23.1 Extreme Value Theory

Reading: Brooks (2019, Section 14.3)

In this section, we are interested in extreme and rare events such as stock price crashes. Therefore, we will look into the left tail of the distribution of returns. We will use the data set provided in **'sp500.xlsx'**, which contains daily returns on the S&P500 index from January 1950 until July 2018. From the price series, we generate log returns by typing

```
sp500$ret = c(NA,diff(log(sp500$sp500)))
sp500 = sp500[-1,]
```

and delete the first row from the data set. As we are focused on the lower tail, the 1%, 5% and 10% percentiles are of interest. They tell us, assuming the historical distribution of returns, what is the maximum loss to expect, with probability 1,5 or 10%. These quantiles can be easily computed with the function **quantile**, which also allows for a list of quantiles. Hence we type

```
quantile(sp500$ret,c(0.01,0.05,0.1))
```

In finance, we also call them Value at Risk (VaR) estimates, e.g., a 1%–VaR of –0.026, as in our case, means that the maximum loss for a day that we can expect equals 2.6%. Since this estimate is derived directly from the historical data, we will also refer to it as the historical VaR. If we were to assume that the returns follow a normal distribution, we can compute the 1%–VaR from the mean (μ) and standard deviation (σ) using the 'delta-normal' approach. The α-percentile of a normal distribution is then given as

$$VaR^{\alpha}_{\text{normal}} = \mu - \Phi^{-1}(1-\alpha)\sigma \tag{17}$$

In R, we simply use the **qnorm** function for this purpose, which will return the requested quantiles for a normal distribution with specific mean and standard deviation. The line of code would be as below.

```
qnorm(c(0.01,0.05,0.1),mean = mean(sp500$ret), sd = sd(sp500$ret))
```

We would then obtain the following output.

```
> quantile(sp500$ret,c(0.01,0.05,0.1))
         1%          5%         10%
-0.02596571 -0.01443812 -0.00988654
> qnorm(c(0.01,0.05,0.1),mean = mean(sp500$ret), sd = sd(sp500$ret))
[1] -0.02214817 -0.01557281 -0.01206751
>
```

We can see that the value $VaR^{1\%}_{\text{normal}} = -0.022$ is smaller in absolute value than the one obtained from the historical distribution. Hence, assuming a normal distribution would lead to the assumption that the maximal loss to expect is only 2.2% but the empirical quantile is further from zero (i.e., more negative, indicating that a larger loss is likely than suggested by the delta-normal approach). However, this is not true for the 5% and 10% quantiles.

23.2 The Hill Estimator for Extreme Value Distributions

Reading: Brooks (2019, Section 14.3)

In the next step, we want to look into the tail of the distribution to estimate the VaR using the Hill estimator for the shape parameter ξ of an extreme value distribution. Below is a short script showing how to create this Hill plot for the VaR, and you can also find the code in the file **'Var_hill_plot.R'**.

```
1  # Script to compute hill plot for value-at-risk
2  load("D:/Programming Guide/R Guide/code/sp500.RData")
3  library(extremefit)
4  U=-0.025; alpha=0.01;
5
6  y = abs(sp500$ret[sp500$ret<U])
7  hill = hill(y)
8  var = sort(-y)*(nrow(sp500)*alpha/length(y))^(-hill$hill)
9
10 # Hill plot of Value-at-Risk
11 par(lwd=2,cex.axis = 2, cex.lab =1.5)
12 plot(var,pch="+",xlab="Order Statistic",ylab="VaR")
13 title("Hill Plot")
```

In the first lines, we import the data set and include the package **extremefit**, which provides a function for the Hill estimates (**hill**). We set the threshold below which we consider returns for the estimation **(U)** and the confidence level **alpha** in line 4. In line 6, we generate the data series $\tilde{y}$, which comprises the absolute value (**abs()**) of all returns below the threshold U. For the Hill estimation, we only need to feed **hill** with this vector and it will return a list of four variables for all order statistics of the input. In this case, the vector **y** has 197 entries. Finally, we compute the VaR in line 8 following Brooks (2019):

$$\text{VaR}_{hill,k} = \tilde{y}_{(k)} \left[\frac{N\alpha}{N_U}\right]^{-\hat{\xi}_k}, \tag{18}$$

where we again make use of vectorisation. Note that we sort the negative values of **y** since we took absolute values before. Applying the whole vector **hill$hill** in the exponents will calculate the VaR for the respective order statistic using the Hill estimator in one step and will write it to the vector **var**.

In lines 11 to 13, we specify the options for the graphic in Figure 26.

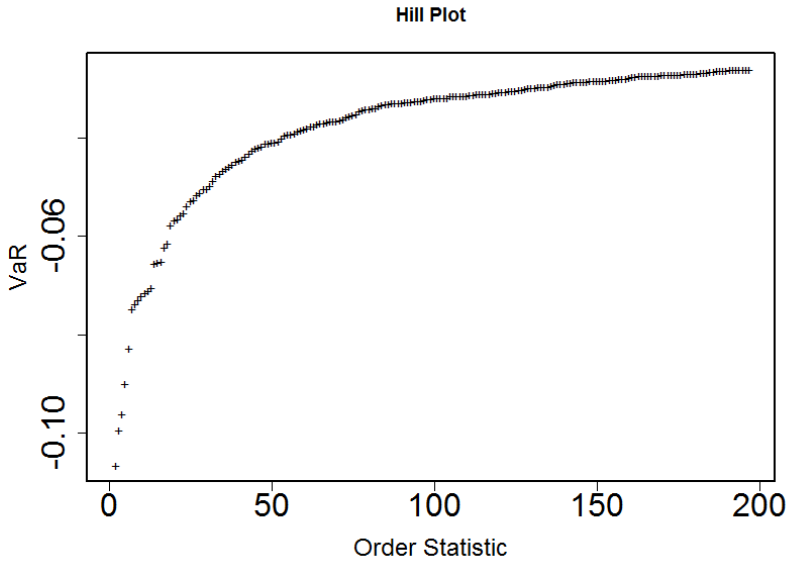


Figure 26: Hill Plot for Value at Risk

Looking at the Hill plot, we can see that the estimated Value-at-Risk is much higher in absolute value. Using all observations, we obtain $VaR_{hill} = -0.032$, which means that the expected maximum is 3.2%.

23.3 VaR Estimation Using Bootstrapping

Reading: Brooks (2019, Section 13.9)

The following R code can be used to calculate the minimum capital risk requirement (MCRR) for a ten-day holding period (the length that regulators require banks to employ) using daily S&P500 data, which is found in the file **'sp500.xlsx'**. The code is presented on the following page followed by comments on some of the key lines.

Again, annotation of the R code will concentrate on commands that have not been discussed previously. The first lines of commands load the dataset and the **rugarch** package necessary for the estimation of univariate GARCH models. In lines 6 and 7, a GARCH(1,1) model with only an intercept in the mean equation is fitted to the return series. The estimated coefficients are written to the variables **mu**, **omega**, **alpha** and **beta** in lines 8 and 9. Line 11 saves the fitted conditional variances in the series **h**. The two lines 12 and 13 will construct a set of standardised residuals **sres**.

Next follows the core of the program, which involves the bootstrap loop (lines 18 to 32). The number of replications has been defined as 10,000. What we want to achieve is to re-sample the series of standardised residuals (**'sres'**) by drawing randomly with replacement from the existing dataset. After defining the three series for the forecasted values for the variance **h_fc**, returns **ret_fc** and the price series **sp500_fc**, in line 19 we use the simple function **sample** to draw a sample of 10 observations with replacement. Although this is a kind of bootstrapping, we do not need to use the R package **boot**, which includes a lot of bootstrapping tools.

Within the next block of commands, the future path of the S&P500 return and price series as well as the conditional variance over the ten-day holding period are computed. This is done in two steps. The first three lines (22 to 24) create the one-step ahead forecasts, before the loop in lines 26 to 30 creates the next nine steps-ahead forecasts.

```
1 # VaR estimation using bootstrapping
2 load("D:/Programming Guide/R Guide/code/sp500.RData")
```

```
3  library(rugarch)
4
5  set.seed(12345)
6  spec = ugarchspec(mean.model = list(armaOrder=c(0,0)),variance.model = list(
       garchOrder=c(1,1),model="sGARCH"))
7  garch11 = ugarchfit(spec,data = sp500$ret)
8  mu = garch11@fit$coef["mu"]; omega = garch11@fit$coef["omega"]
9  alpha = garch11@fit$coef["alpha1"]; beta = garch11@fit$coef["beta1"]
10
11 h = garch11@fit$var
12 resid  = (garch11@fit$residuals - mu)
13 sres = resid/h^0.5
14
15 N = length(h)
16 mcrr = NULL
17
18 for (n in 1:1000) {
19   h_fc = ret_fc = sp500_fc = NULL
20   random = sample(sres,size = 10,replace = T)
21   # Constructing one ste ahead forecast
22   h_fc = omega + alpha*resid[N]^2 + beta*h[N]
23   ret_fc = mu + sqrt(h_fc[1])*random[1]
24   sp500_fc = sp500$sp500[N]*exp(ret_fc)
25   # Loop for the next 9 step ahead forecasts
26   for (i  in 1:9){
27     h_fc = c(h_fc, omega+(alpha+beta)*h_fc[i])
28     ret_fc = c(ret_fc,mu+sqrt(h_fc[i+1])*random[i+1])
29     sp500_fc = c(sp500_fc,sp500_fc[i]*exp(ret_fc[i+1]))
30   }
31   mcrr = rbind(mcrr, log(c(min(sp500_fc),max(sp500_fc))/sp500$sp500[N]))
32 }
33
34 mcrrl = 1 - exp(mean(mcrr[,1]) - 1.645*sd(mcrr[,1]))
35 mcrrs = exp(mean(mcrr[,2]) + 1.645*sd(mcrr[,2])) - 1
```

Recall from Brooks (2019) that the one-step-ahead forecast of the conditional variance is

$$h_{1,T}^f = \alpha_0 + \alpha_1 u_T^2 + \beta h_T, \tag{19}$$

where h_T is the conditional variance at time T, i.e., the end of the in-sample period, and u_T is the squared disturbance term at time T, and α_0, α_1 and β are the coefficient estimates obtained from the GARCH(1,1) model estimated over the observations 2 to T. The s-step-ahead forecast can be produced by iterating

$$h_{s,T}^f = \alpha_0 + (\alpha_1 + \beta) h_{s-1,T}^f \qquad s \geq 2. \tag{20}$$

Following the above formula in Equation (19), we compute the one-step ahead forecast for the variance in line 22. From this, we obtain a return forecast by multiplying the square root (**sqrt**) of the variance with the error drawn from the standardised residuals and adding the mean intercept. Finally, with the return forecast we can determine the respective price forecast by compounding. Once we have determined the one-step ahead forecasts, the next steps are obtained recursively. Therefore, we use the loop in line 26

to 30 and consecutively update the variance, return and price series.[36] Finally, we obtain the minimum and maximum values of the S&P500 series over the ten days and compute the holding period return as the log of the fraction with the last price and post them into the **'mcrr'** file.

This set of commands is then repeated 10,000 times so that after the final repetition there will be 10,000 minimum and maximum returns for the S&P500. The final block of commands generates the MCRR for a long and a short position in the S&P500. We now want to find the 5th percentile of the empirical distribution of maximum losses for the long and short positions. Under the assumption that the statistics are normally distributed across the replications, the MCRR can be calculated by the commands in lines 34 and 35, respectively. The results generated by running the above program should be displayed in the *Console* window and should approximate to

`mcrrl` = 0.03467963 for the long position, and

`mcrrs` = 0.03281008 for the short position.

These figures represent the minimum capital risk requirement for long and short positions, respectively, as percentages of the initial value of the position for 95% coverage over a 10-day horizon. This means that, for example, approximately 3.5% of the value of a long position held as liquid capital will be sufficient to cover losses on 95% of days if the position is held for 10 days. The required capital to cover 95% of losses over a 10-day holding period for a short position in the S&P500 index would be around 3.3%. Higher capital requirements are thus necessary for a long position since a loss is more likely than for a short position of the same magnitude.[37]

[36]Note that the order of the commands is important as the newly assigned conditional variance **h** in line 27 is used in line 28 to update the return series, which then is used to update the index level in line 29.

[37]Note that the estimation can be quite different depending on the software package used, since from the outset the GARCH(1,1) coefficient estimates can differ and hence will influence the whole result.

24 The Fama–MacBeth Procedure

Reading: Brooks (2019, Section 14.2)

In this section, we will perform the two-stage procedure by Fama and MacBeth (1973). The Fama–MacBeth procedure, as well as related asset pricing tests, are described in detail in Brooks (2019). There is nothing particularly complex about the two-stage procedure – it only involves two sets of standard linear regressions. The hard part is really in collecting and organising the data. If we wished to do a more sophisticated study – for example, using a bootstrapping procedure or using the Shanken (1992) correction, this would require more analysis then is conducted in the illustration below. However, hopefully the R code and the explanations will be sufficient to demonstrate how to apply the procedures to any set of data.

The example employed here is taken from the study by Gregory et al. (2013) that examines the performance of several different variants of the Fama and French (1992) and Carhart (1997) models using the Fama–MacBeth methodology in the UK following several earlier studies showing that these approaches appear to work far less well for the UK than the US. The data required are provided by Gregory et al. (2013) on their web site.[38] Note that their data have been refined and further cleaned since their paper was written (i.e., the web site data are not identical to those used in the paper) and as a result, the parameter estimates presented here deviate slightly from theirs. However, given that the motivation for this exercise is to demonstrate how the Fama–MacBeth approach can be used in R, this difference should not be consequential.

The two data files used are **'monthlyfactors.xlsx'** and **'vw_sizebm_25groups.xslx'**. The former file includes the time series of returns on all of the factors **(smb, hml, umd, rmrf)**, the return on the market portfolio **(rm)** and the return on the risk-free asset **(rf)**), while the latter includes the time series of returns on 25 value-weighted portfolios formed from a large universe of stocks, two-way sorted according to their sizes and book-to-market ratios.

The steps to obtain the market prices of risk and the respective t-statistics is set out in the R script **fama_macbeth.R** and are displayed on the next page. The first step in this analysis for conducting the Fama–French or Carhart procedures using the methodology developed by Fama and MacBeth (1973) is to create a joint data set form the two files. The data in both cases run from October 1980 to December 2017, making a total of 447 data points. However, in order to obtain results as close as possible to those of the original paper, when running the regressions, the period is from October 1980 to December 2010 (363 data points).

We need to transform all of the raw portfolio returns into excess returns, which are required to compute the betas in the first stage of Fama–MacBeth. This is fairly simple to do and we just write over the original series with their excess return counterparts in line 6. Note that we have to exlude the date variable though, and therefore add the **[,-1]** after the portfolios. After putting both datasets into a joint data frame called **data**, we adjust this data frame to include only the first 363 observations in order to ensure that the same sample period as the paper by Gregory et al. (2013) is employed throughout.

Next, we run the first stage of the Fama–MacBeth procedure, i.e., we run a set of time-series regressions to estimate the betas. We want to run the Carhart (1997) 4-factor model separately for each of the twenty-five portfolios. A Carhart 4-factor model regresses the portfolio returns on the excess market returns (**'rmrf'**), the size factor (**'smb'**), the value factor (**'hml'**) and the momentum factor (**'umd'**).

Since the independent variables remain the same across the set of regressions and we only change the dependent variable, i.e., the excess return of one of the 25 portfolios, we can set this first stage up as a loop. Lines 12 to 16 specify this loop, while we initialise the variable **betas** in line 12 to store the β estimates in later. To run through all the portfolios, set the loop up to run from 2 to 26, which denote

[38] `http://business-school.exeter.ac.uk/research/centres/xfi/famafrench/files/`

the 25 columns containing the portfolios in the new dataset **data**. The time-series regressions in line 14 then always consist of the same independent variables from the dataset and changing portfolios on the left hand side. In line 15, coefficients 2 to 5 are extracted and appended to the variable beta. Note that we do not need to save the intercepts and hence exclude the first entry.

```
1  # Fama MacBeth two stage procedure
2  library(readxl)
3  # Read data and create joint data set
4  monthlyfactors <- read_excel("D:/Programming Guide/data/monthlyfactors.xlsx"
       )
5  vw_sizebm_25groups <- read_excel("D:/Programming Guide/data/vw_sizebm_25
       groups.xlsx")
6  vw_sizebm_25groups[,-1] = vw_sizebm_25groups[,-1] - monthlyfactors$rf
7  data = data.frame(vw_sizebm_25groups,monthlyfactors[,2:7])
8  data = data[1:363,] # Adjust data set to Gregory et al. period
9
10 # First stage regressions
11 betas = NULL
12 for (var in 2:26) {
13   lr = lm(data[,var] ~ rmrf + smb + hml + umd, data = data)
14   betas = rbind(betas,lr$coefficients[2:5])
15
16 }
17
18 # Second stage regressions
19 lambdas = Rsq = NULL
20 for (t in 1:nrow(data)) {
21   lr = lm(t(data[t,2:26]) ~ betas)
22   lambdas = rbind(lambdas, lr$coefficients)
23   Rsq = c(Rsq,summary(lr)$r.squared)
24 }
25
26 # Compute market prices of risk and t-statistics
27 colMeans(lambdas)*100
28 nrow(data)^0.5*colMeans(lambdas)/apply(lambdas, 2, sd)
29 mean(Rsq)
```

Having run the first step of the Fama–MacBeth methodology – we have estimated the betas, also known as the factor exposures. The slope parameter estimates for the regression of a given portfolio will show how sensitive the returns on that portfolio are to the corresponding factors. We did not save them here, but the intercepts ('**lr$coefficients[1]**') would be the Jensen's alpha estimates. These intercept estimates should be comparable to those in the second panel of Table 6 in Gregory et al. (2013) – their column headed '*Simple 4F*'. Since the parameter estimates in all of their tables are expressed as percentages, we need to multiply the figures by 100 for comparison.

If the 4-factor model is a good one, we should find that all of these alphas are statistically insignificant. We could test them individually, if we wished, by adding an additional line of code in the loop to save the t-ratio in the regressions. However, we avoid this here, since we will be working with the β estimates only.

For the second stage regressions, we set up a similar loop from line 20 to 24, again initialising the variable **lambdas** for storing the results as before. Since these will be cross-sectional regressions, the

loop variable **t** is now running through the rows of the dataset (from 1 to 363 for the adjusted dataset). The statements also again only change in the dependent variable between the loops, as we always use the same betas for the regressions. Since the matrix betas include all necessary independent variables, we do not need to split them up, but R does this automatically if we provide a matrix as an independent variable.

The second stage cross-sectional regressions correspond to the following equation

$$\bar{R}_i = \alpha + \lambda_M \beta_{i,M} + \lambda_S \beta_{i,S} + \lambda_V \beta_{i,V} + \lambda_U \beta_{i,U} + e_i \tag{21}$$

This is performed in line 21. We also collect the R^2 values for each regression in the list **Rsq** as it is of interest to examine the cross-sectional average. We store all coefficients in the **lambdas** and the R^2 values in **Rsq**. Note that the R^2 can only be accessed after applying the **summary** function to the linear model.

The final stage of the Fama–MacBeth procedure is to compute the averages, standard deviations and t-ratios from the series of estimates from the second stage. For every factor j, we compute the time-series averages and standard deviations of the estimates $\hat{\lambda}_{j,t}$ as follows

$$\hat{\lambda}_j = \frac{1}{T}\sum_{t=1}^{T} \hat{\lambda}_{j,t}, \quad \hat{\sigma}_j = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(\hat{\lambda}_{j,t} - \hat{\lambda}_j)^2} \quad \text{and} \quad t_{\lambda_j} = \sqrt{T}\,\frac{\hat{\lambda}_j}{\hat{\sigma}_j} \tag{22}$$

To compute the means, we can use the function **colMeans** which is a vectorisation of **mean** to be applied to the columns of a matrix. Unfortunately, there is no such function for the standard deviation, but we can create it directly using **apply**. The command **apply(lambdas,2,sd)** applies the function **sd** to the second dimension, i.e., the columns of the matrix of lambdas. In this way, we obtain the column-wise standard deviations and can compute the t-statistics as given in Equation (22) directly in one line.

The lambda parameter estimates should be comparable to the results in the columns headed '*Simple 4F Single*' from Panel A of Table 9 in Gregory et al. (2013). Note that they use γ to denote the parameters which have been called λ in this guide and in Brooks (2019). The parameter estimates obtained from this simulation and their corresponding t-ratios are given in the table below. Note that the latter do not use the Shanken (1992) correction as Gregory et al. (2013) do. These parameter estimates are the prices of risk for each of the factors, and interestingly only the price of risk for value is significantly different from zero.

Table 3: Fama–MacBeth Market Prices of Risk

Parameter	Estimate	t-ratio
λ_C	0.50	1.33
λ_{RMRF}	0.07	0.15
λ_{SMB}	0.09	0.51
λ_{HML}	0.39	2.08
λ_{UMD}	0.41	0.78
Mean R^2	0.39	

References

Anscombe, F. J. and Glynn, W. J. (1983). Distribution of the kurtosis statistic b 2 for normal samples. *Biometrika*, 70(1):227–234.

Brooks, C. (2019). *Introductory Econometrics for Finance*. Cambridge University Press, Cambridge, UK, 4th edition.

Carhart, M. M. (1997). On persistence in mutual fund performance. *Journal of Finance*, 52(1):57–82.

D'Agostino, R. B. (1970). Transformation to normality of the null distribution of g1. *Biometrika*, 57(3):679–681.

Durbin, J. and Watson, G. S. (1951). Testing for serial correlation in least squares regression. ii. *Biometrika*, 38(1/2):159–177.

Elliot, B., Rothenberg, T., and Stock, J. (1996). Efficient tests of the unit root hypothesis. *Econometrica*, 64(8):13–36.

Fama, E. F. and French, K. R. (1992). The cross-section of expected stock returns. *Journal of Finance*, 47(2):427–465.

Fama, E. F. and MacBeth, J. D. (1973). Risk, return, and equilibrium: Empirical tests. *Journal of Political Economy*, 81(3):607–636.

Fuller, W. A. (1976). *Introduction to Statistical Time Series*, volume 428. John Wiley and Sons.

Gregory, A., Tharyan, R., and Christidis, A. (2013). Constructing and testing alternative versions of the fama–french and carhart models in the uk. *Journal of Business Finance and Accounting*, 40(1-2):172–214.

Haug, E. G. (1998). *The Complete Guide to Option Pricing Formulas*. McGraw-Hill New York.

Heslop, S. and Varotto, S. (2007). Admissions of international graduate students: Art or science? a business school experience. *ICMA Centre Discussion Papers in Finance*, 8.

Koenker, R. (1994). Confidence intervals for regression quantiles. In *Asymptotic statistics*, pages 349–359. Springer.

Nelson, D. B. (1991). Conditional heteroskedasticity in asset returns: A new approach. *Econometrica*, 59(2):347–370.

Shanken, J. (1992). On the estimation of beta-pricing models. *Review of Financial Studies*, 5(1):1–33.

White, H. (1980). A heteroskedasticity-consistent covariance matrix estimator and a direct test for heteroskedasticity. *Econometrica*, 48(4):817–838.