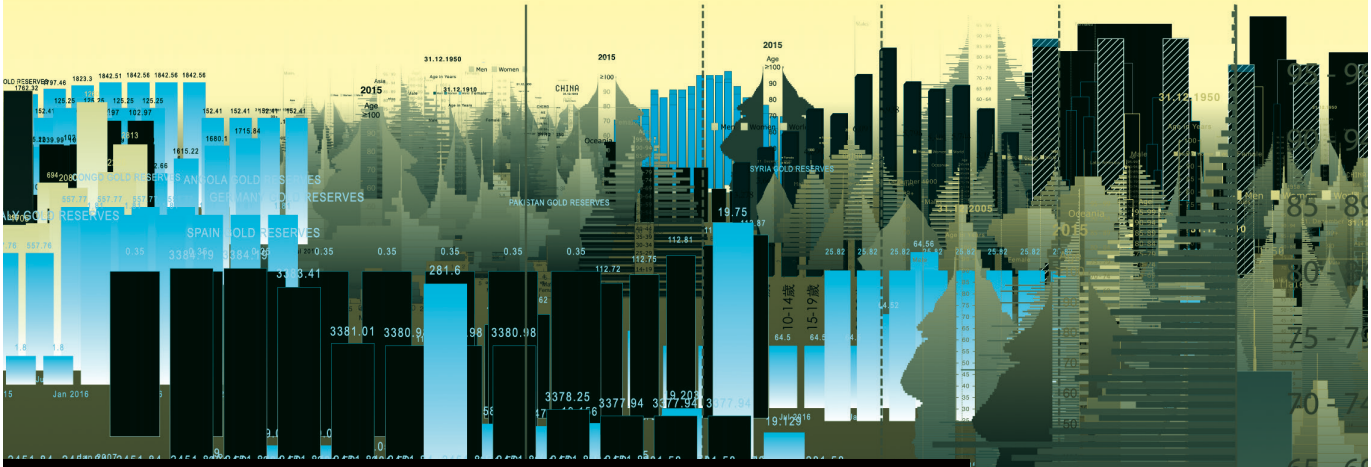
RAN TAO
CHRIS BROOKS



PYTHON 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.

Contents

List of Figures

List of Tables

1 Getting started

1.1 What is Python?

Python is an open-source, high-level and interpreted programming language for various purposes. Several libraries it provides have proved particularly useful for both financial industry practitioners and academic researchers, and therefore the programming language has become increasingly popular.

Python has several strengths: first, it is user-friendly for programming beginners. Learners do not need to spend too much time on its syntax because the language is simple and very concise, meaning that the code is highly efficient and much can be achieved in just a few lines compared with other languages. Second, it provides extensive support libraries, web service tools, string operations, panel data analysis, etc. Most frequently used programming tasks have already been scripted and users can easily import them for their own purposes. Finally, it is free of charge and thus open to everyone. Users can easily step into statistical analysis by implementing code in Python.

Most relevant for our purposes, Python can also implement many different statistical and econometric tests. StatsModels is a powerful Python library to conduct many statistical tests, and in this guide it will be used frequently.

Note that this guide is based on Python version 3.6. However, there are significant differences between version 2.7 and versions 3.5 and above. If you use the earlier version of Python (2.7), the syntax as well as certain modules it provides are different. Moreover, as Python is an open-source programming language and has many libraries or packages updated frequently, some functions might no longer be supported or may have moved to other modules in future versions.

This section assumes that readers have successfully set up an environment and downloaded the Python software onto an available computer. There are a number of ways to implement Python code. Note that this book implements all example code in Jupyter NoteBook via Anaconda.[1] Alternatively, you can choose PyCharm, Spyder and etc. dependent on your own preferences. Spyder, for example, is also a popular Python environment that is available through Anaconda, but it is not interactive, meaning that you write the code and then run it with the output appearing in a different window. There are no differences between these environments in terms of the syntax itself, however.

A good way of familiarising yourself with Python is to learn about its syntax and go through the examples given in this guide. There now follows a presentation of the fundamentals of the Python package, together with instructions on how to achieve standard tasks and sample output. All example code is given, with different colours highlighting different parts of the code. A number of comments and explanations along with this code will hopefully make it easier to understand. Additionally, it is noteworthy that Python has a strict rule for naming variables. For example, variable names must start with a letter or underscore, any phrase variable must be integrated by an underscore instead of adding blank in between, and it is also CASE-SENSITIVE. Thus, it is important to enter commands exactly as the example shows in order to avoid unnecessary errors.

Finally, it is worth mentioning that the aim of this book is to help finance students and researchers who are new to Python to quickly master the basic syntax and knowledge to enable them to implement various applications of econometric tests. Since the core of Python is not a specialist econometric programming language like R or STATA, some statistical output looks very preliminary and a number of econometric tests are not yet developed or tested in Python. To cover as many examples from *Introductory Econometrics for Finance* as possible, we attempt to design some functions to complete these tests. Undoubtedly, there is still considerable room to improve the code and no doubt the outcomes could have been achieved in different ways.

[1] Jupyter is an open-source, web-based, interactive tool for implementing the Python language. You can either download Jupyter directly or access it by installing the Anaconda platform.

1.2 Different ways to run Python code

Unlike other statistic software, there are a number of ways to run Python code. The simplest way to write and execute Python code is via the Python console. Assuming that the reader has installed Python on a Windows-based computer, to open the console/terminal, we click on **Start** and type 'cmd' in the search engine. Then hitting **ENTER** leads the Windows system to launch a new interface (Figure 1). Next, we type 'Python' and press **ENTER**, in which case the Python console opens. We can now directly write and execute code here.

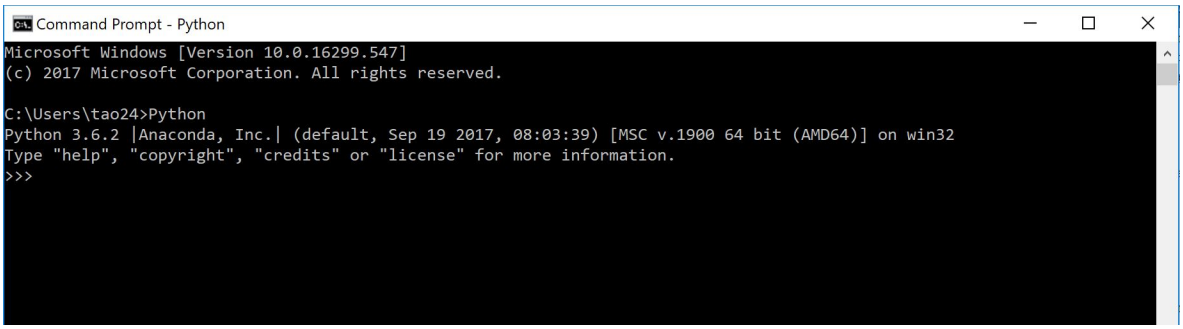


Figure 1: Opening the Python Console

Alternatively, you can write your code in an editor and execute it from the console. For example, we can first open a Notepad++ file and write the command line **print("Hello World")**, saving the file as name of **program.py**. Then we repeat the procedure as stated above. Click **Start**, search for 'cmd' and press **ENTER**. You need to navigate the Python console to where you have saved the file (in this case, we have saved the file on the Desktop). Write the following line to execute the programme (Figure 2).

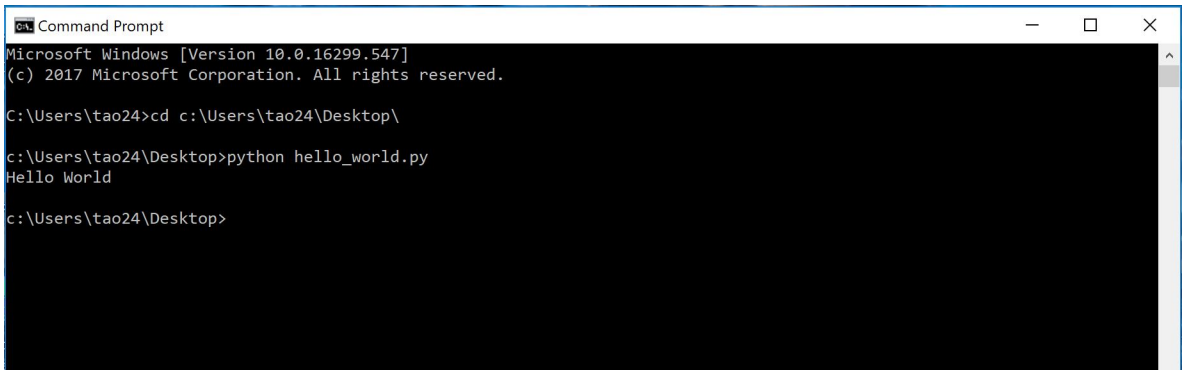


Figure 2: Running Code from the Console

However, it is usually more convenient to implement Python code using an IDE (Integrated Development Environment). In this book, we use 'Anaconda', which will first need to be downloaded.[2] Once the software is on the computer, click on **Start**, search for 'Anaconda' and open the programme 'Anaconda Navigator'. As you can see in Figure 3, Anaconda is a collection of software packages. There are several applications that can be launched such as the Jupyter NoteBook, Spyder, etc. For

[2] Anaconda can be downloaded from https://www.anaconda.com/download/

the purpose of better presenting code in this book, we choose the Jupyter NoteBook because it is designed as an open-source Web-based, interactive computing application.[3]

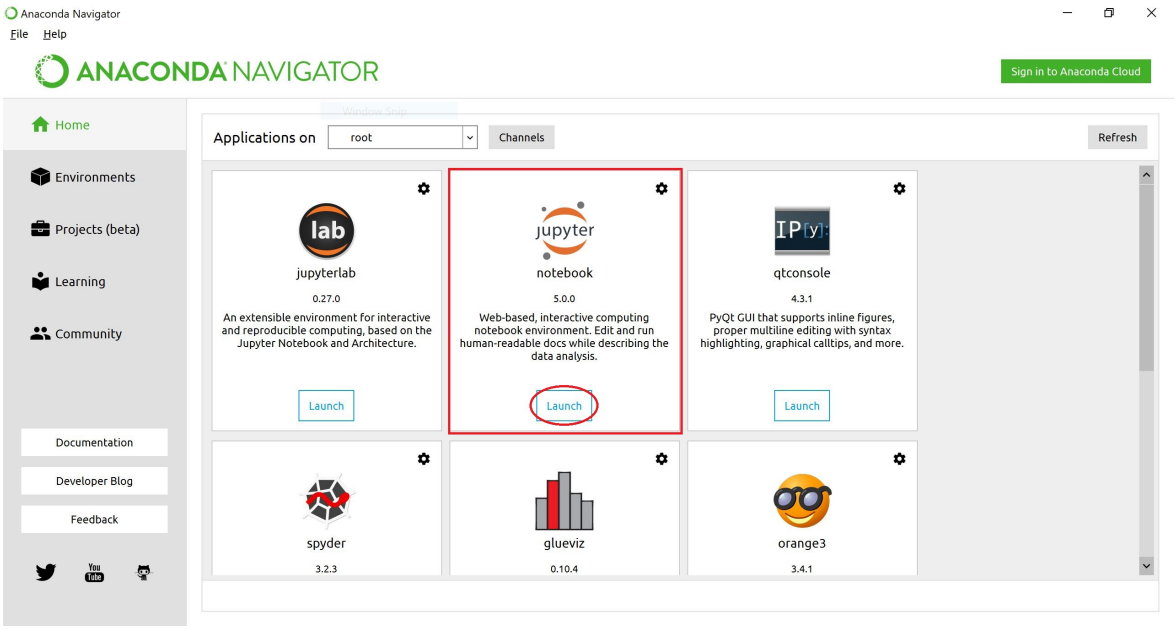


Figure 3: Opening the Anaconda Navigator

We now step into how to use Jupyter NoteBook. To launch this application, click the **Launch** button below the icon (highlighted by a red circle). Anaconda then takes a few seconds to open a web page (Figure 4). You can see a number of folders on the main screen. Choose a folder as the directory for the Python code. For instance, we can click on the folder **Desktop** and then create a new script by clicking the button **New**. In the drop-down menu, click **Python 3**. Jupyter then creates a new web page where the code can be written and executed. The detail of how to use Jupyter will be presented in the next section.

[3]Note that readers need to download Anaconda with caution since, as stated above, there are two different versions. One is for Python 2.7, and the other is for Python 3.x. We will mainly focus on the Python 3.x version.

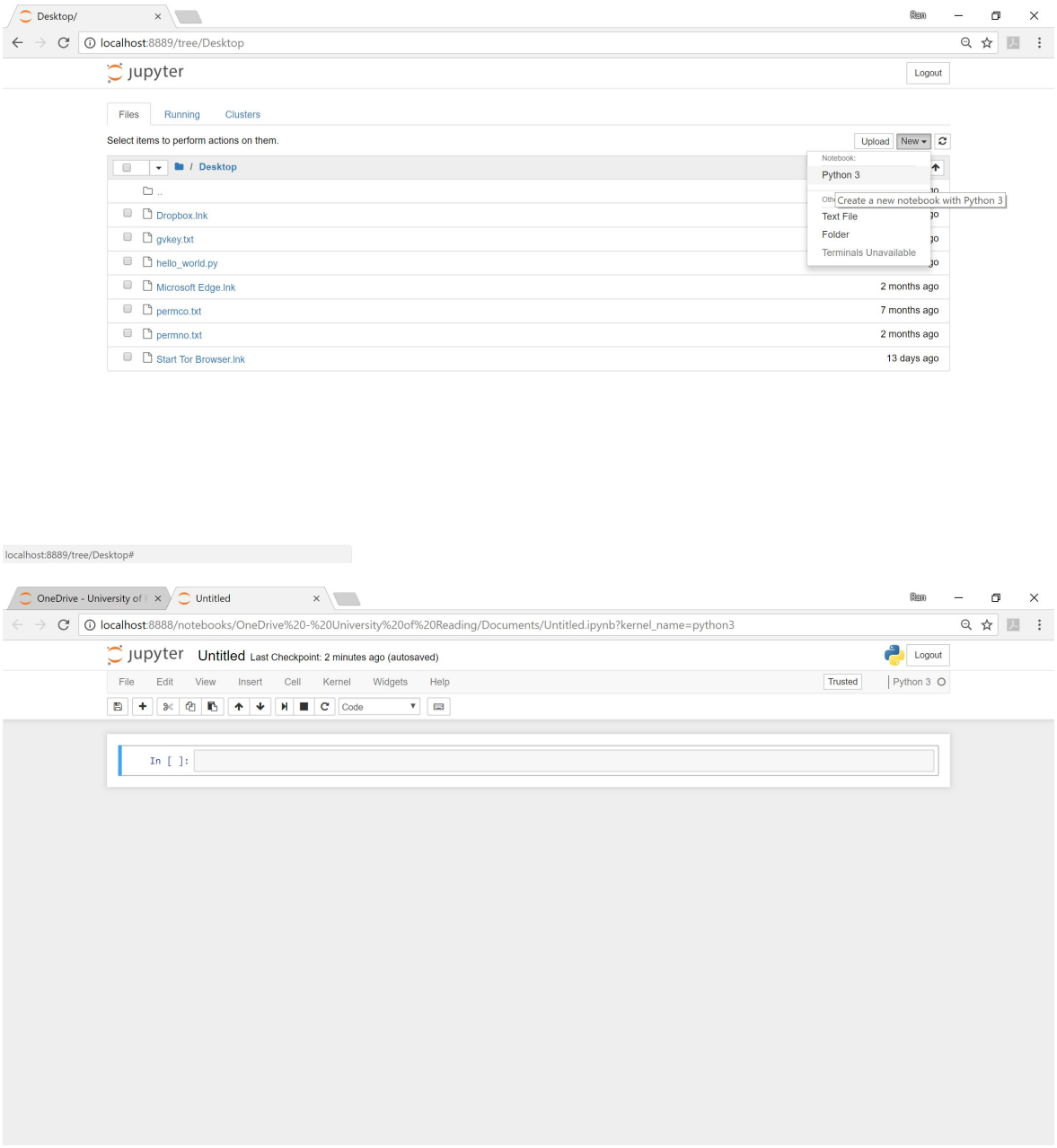


Figure 4: Opening a Jupyter NoteBook

1.3 What does a Jupyter NoteBook look like?

Let us briefly introduce the Jupyter NoteBook interface. Once a NoteBook has been opened, you will immediately see the NoteBook name, a menu bar, a tool bar and an empty code cell. As can be seen in Figure 5, each area has been highlighted by different colours.

Firstly, you can rename the NoteBook by hitting the current file name **Untitled**. Jupyter then opens a new window where you can revise the name (see Figure 6). For example, we rename the NoteBook 'hello_world' and click the 'rename' button. You can now see the new name in the top left corner.

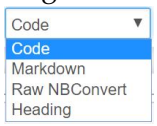
The tool bar lying between the menu bar and the code cell offers some functions frequently used in the NoteBook. The first icon  allows you to quickly save the NoteBook. The next button can add an additional code cell right below the existing one. Then, the three consecutive icons  provides the function of cut, copy and paste code cells, respectively.  moves the selected code cell up and down. Moreover,  are the functions for running selected code cell, interrupting selected code cell and restarting the kernel from left to right. There is also a drop-down menu  where you can define the cell type as code, Markdown, Raw NBConvert or Heading.[4]

Figure 5: The Interface of a Jupyter NoteBook

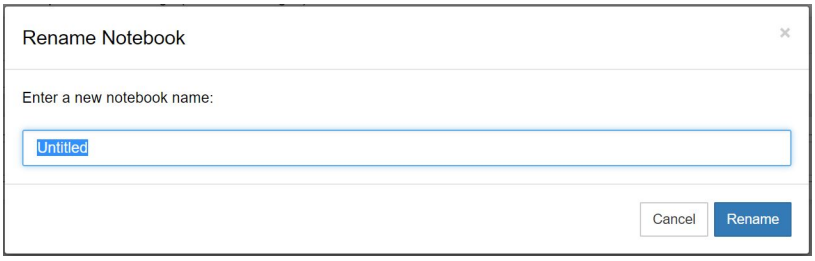


Figure 6: Renaming Jupyter NoteBook

[4]The specific functions of these terms is explained as follows: code is used for writing Python code; Markdown allows you to edit cells using plain language, formulae and etc.; Raw NBConvert prevents the cell from being executed; Heading is used to set up titles.

Option	**Purpose**
New NoteBook	Create a new NoteBook
Open...	Create a new NoteBook Dashboard
Make a Copy...	Copy the current NoteBook and paste it into a new NoteBook
Rename...	Change the name of the current NoteBook
Save and Checkpoint	Save the NoteBook as a checkpoint
Revert to Checkpoint	Revert the NoteBook back to a saved checkpoint
Print Preview	Print Preview
Download as	Download the NoteBook as certain type of file
Close and Halt	Stop running and exit the NoteBook

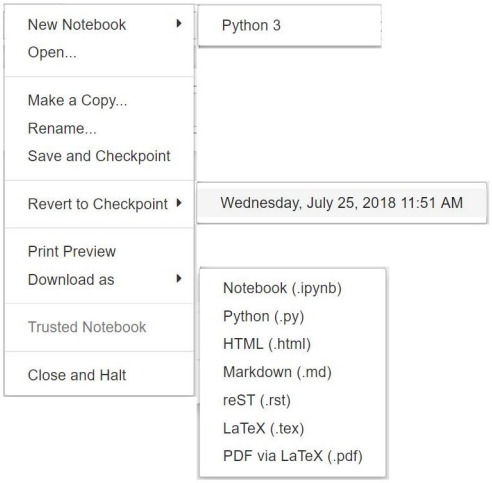


Figure 7: File Menu Options

The menu bar contains **File**, **Edit**, **View**, **Insert**, **Cell**, **Kernel**, **Help** functions. We will now go through each of them individually.

As can be seen in Figure 7, there are a number of options that can be selected. The purpose of each option is explained as follows:

Option	Purpose
Cut Cells	Cut the selected cell
Copy Cells	Copy the selected cell
Paste Cells Above	Paste cell above the current cell
Paste Cells Below	Paste cell below the current cell
Paste Cells & Replace	Paste cell into the current cell
Delete Cells	Delete the selected cell
Undo Delete Cells	Undo the delete action
Split Cell	Split cell given the position of mouse cursor
Merge Cell Above	Merge cell with the one above
Merge Cell Below	Merge cell with the one below
Move Cell Up	Bring up cell
Move Cell down	Bring down cell
Edit NoteBook Metadata	Edit NoteBook metadata
Find and Replace	Find and replace the target code or text

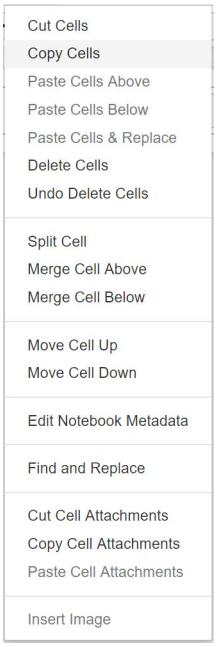


Figure 8: Edit Menu Options

The **Edit** button gives a drop-down menu where a number of options are listed (see Figure 8). The purpose of each option is explained in the following table.

Option	**Purpose**
Toggle Header	Hide or display the logo and name of the Jupyter NoteBook
Toggle Toolbar	Hide or display the tool bar
Cell Toolbar	Change the type of cell for different form of presentation

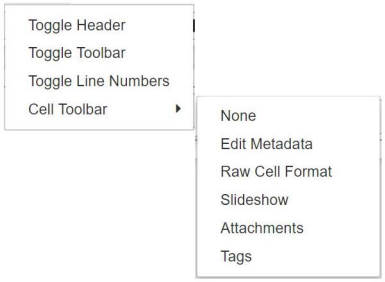


Figure 9: View Menu Options

The **View** menu (see Figure 9) provides some functions for changing the layout of a Jupyter NoteBook. Note that this menu has no impact on the code itself.
The **Insert** menu looks straightforward, with only two options 'Insert Cell Above' and 'Insert Cell Below'. If you want to add an additional cell among the existing cells, this is the method you can employ. Alternatively, the button + on the tool bar can be used for the same purpose.

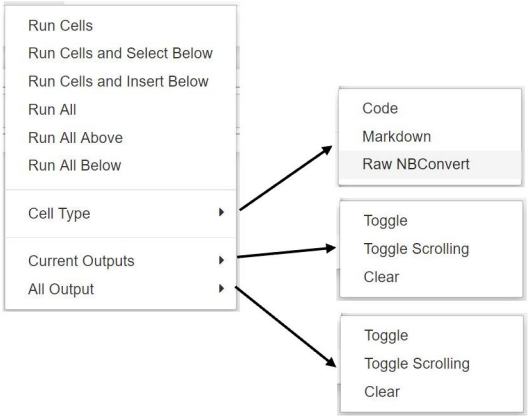


Figure 10: Cell Menu Options

The menu next to **Insert** is the **Cell** menu. Figure 10 reports all options that it contains and each function is explained in the following table.

Option	Purpose
Run Cells	Execute the code from selected cell
Run Cells and Select Below	Execute selected cell and move to next cell
Run Cells and Insert Below	Execute selected cell and insert a new cell below
Run All	Execute all cells
Run All Above	Execute all cells above the selected one
Run All Below	Execute all cells below the selected one
Cell Type	Choose the type of cell as code, markdown or Raw NBConvert
Current Output	Hide/Display/Clear the current output of selected cell
All Output	Hide/Display/Clear all output

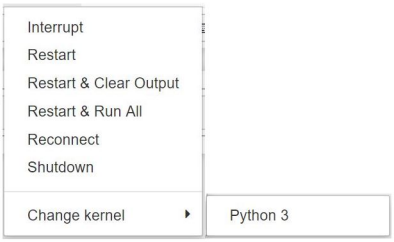


Figure 11: Kernel Menu Options

The **Kernel** menu (Figure 11) contains all necessary kernel actions. For example, we sometimes want to exit the running code and therefore we need to click on the 'Interrupt' option. This is equivalent to the keyboard shortcut **CONTROL** and **C**. The option 'Restart' refers to reboot the kernel. 'Restart & Clear Output' adds an additional function where it can erase all output apart from rebooting the kernel. Moreover, if you want to execute all code cells from the very beginning, 'Restart & Run All' can achieve this. Finally, the last two options, 'Reconnect' and 'Change Kernel', work as their names suggest.

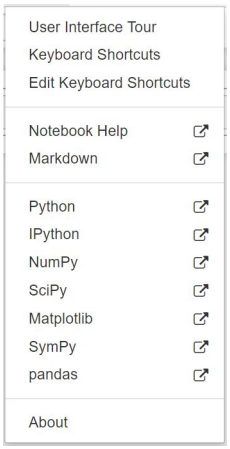


Figure 12: Help Menu Options

Finally, if you are new to Jupyter NoteBook, the **Help** menu (Figure 12) provides a useful 'User Interface Tour', which introduces each function and its options in detail. 'Keyboard Shortcuts' shows all of the shortcuts that you might need for convenience. In addition, several links bring you to

additional web pages such as 'NoteBook Help', 'Markdown', 'Pandas'. If you are unfamiliar with some of the libraries, these websites might offer help.

Let us focus on cells now. You can add, remove or edit code cells according to your requirements. We will run a simple demonstration. Type the code **print("hello world!")**. You will notice that the code cell has a green border on the left side. Meanwhile, a small pencil icon can be observed in the top right corner (see the red circle from Figure 13). This means Jupyter is now in edit mode. Then hitting **SHIFT** and **ENTER** leads Jupyter to execute the code (Alternatively, the code can be run by clicking the button  on the tool bar, or by going to the **Cell** menu and clicking **Run Cells**).

You can see the text output, "hello world!" right below the code cell. Additionally, the left border becomes blue and the icon pencil disappears. These changes tell you that Jupyter has turned to command mode and the code cell cannot be edited unless the user double-clicks it, getting back to edit mode. Furthermore, it can be observed that the icon on the left side of the code cell changes from 'In [ ]:' to 'In [1]:'. The number inside the square bracket refers to a serial number of the code execution. If you see the 'In [*]' symbol, it means that the code is running.

Recall that there are four cell types: Code, Markdown, Raw NBConvert and Heading, which can be converted easily by either typing shortcuts or by clicking menu options. In other words, it is evident that the cell cannot only edit Python code but it also supports the insertion of explanatory text, titles, subtitles, formula or even images to clarify the code. This makes a Jupyter NoteBook a real notebook in the end.

For example, let us type a sentence, "This is the first Jupyter tutorial" in a empty cell. Then go to the drop-down menu on the tool bar and click **Markdown**. You will see the colour of the sentence becomes blue and a '#' symbol appears at the very beginning. Now hit **SHIFT** and **ENTER** simultaneously. The sentence changes to a black and bold title. Note that a single '#' represents a first level heading. Two '#'s leads the heading to level two. By adding more '#', the text will decrease to correspondingly lower levels. Moreover, formulae can be presented in cells. Fortunately, Jupyter supports the Markdown language and is consistent with Latex formatting.[5] For instance, type "$ y = x ^ 2 $" in a empty cell and press **SHIFT** and **ENTER**. Jupyter then produces a mathematical formula "$y = x^2$".

[5]Latex is a document presentation system where plain text is required. However, this book is not the place to learn how to write and produce Latex documents. More advanced usage of the Markdown language such as how to insert images will not be demonstrated in detail. If readers are interested in the presentation of Latex files, websites such as https://www.sharelatex.com/learn/ might be helpful.

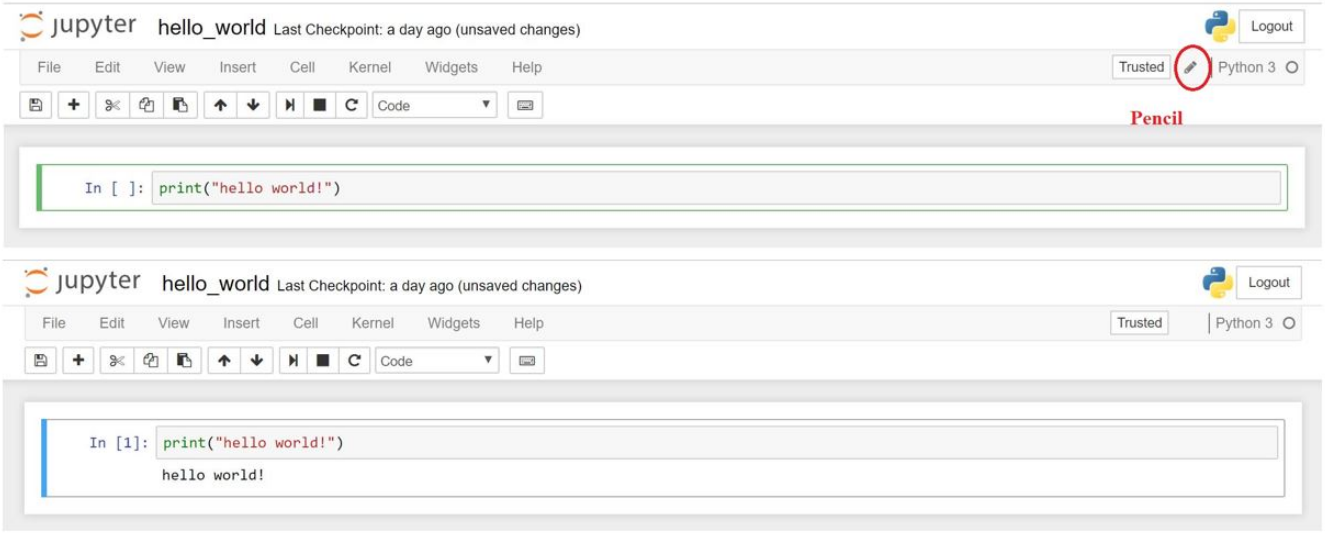


Figure 13: The Cell Edit and Command Mode

1.4 Getting help

There are several different ways to get help when using Python. Firstly, If you are using a Jupyter NoteBook, it has a **Help** menu, which provides several web links to explore some of Python's common libraries – namely, NumPy, Scipy, Pandas, etc. This documentation offers very detailed insights into each method, function, attribution or argument. Taking Pandas as an example, the documentation contains numerous tutorials written by different people. Some of them provide concrete examples for getting started with Pandas, for example.

Sometimes you might need help regarding a particular Python function or method. For every function or method, the Jupyter NoteBook has built-in magic commands can be called by typing **?** followed by the specific function. The information you will receive is an abbreviated version of the function manual, which lists every parameter and return in detail.

For a more comprehensive search, or if you do not know what specific function/method to use, we recommend you use a search engine by typing Python followed by the specific keywords. Since Python is an open-source programming language, there are vast resources that may provide support, such as question-answer posts asking for similar help on Stack Overflow, or user-written commands regarding your particular enquiry from Github. Among these two on-line communities, Stack Overflow is highly recommended. This free on-line programming community is a great source if you have questions regarding specific functionalities or you do not know how to implement particular requirements in Python. Browsing questions and answers written by the community is available without registration; if you wish to post questions and/or provide answers to a posted question, you would need to register first.[6]

[6]https://stackoverflow.com/

2 Data management in Python

Before we step into the application of econometric tests, it is necessary to learn a bit about the Python syntax, such as variable naming rules, when to use a whitespace, how to add comments, doing maths, creating a user-defined functions and so on.

2.1 Variables and name rules

Data type

Any application cannot be processed by Python without storing and working with different types of data. The fundamental way of doing this is first by creating a variable. A variable has to have a name, which is later used to store data. For example, type the following code and hit **SHIFT** and **ENTER**. Once executed, the variable **a** now stores the number 1.

It is useful to know the type of variable. In this case, the variable stores an integer, written in short as **int** in Python. We can check the variable type by keying the command **type()** followed by the variable's name **a** inside the bracket.

```
In [1]: a = 1

In [2]: type(a)

Out[2]: int
```

It is common that finance researchers will be dealing with more precise numbers, i.e., a number followed by a decimal point and digits. This data type is called a **float**. For example, let us type the command **a = 1.2** and **type (a)**. Jupyter will output **float** after executing the cell.

```
In [3]: a = 1.2
        type(a)

Out[3]: float
```

Another data type is 'boolean'. A boolean can only take two values, either **True** or **False**. Let us type the command **b = True** and **c = False**. Then turn the code cells into command mode. Again, typing the command **type (b)** or **type (b)** in the next empty cell will return the data type **bool**, which is the short name for boolean.[7]

```
In [4]: b = True
        c = False

In [5]: type(b)

Out[5]: bool

In [6]: type(c)

Out[6]: bool
```

[7] The value **True** and **False** changes to green colours after execution. This is because these two values are set as Python keywords and have particular purposes. Note that it is not appropriate to use Python keywords as the names of variables since Python will mix up these terms.

Additionally, we may sometimes work on string data instead of numeric data. This is more likely to be the case when we are working with time-series data (a series of numeric data with corresponding datetime stamps). These datetime points will be display as string values. Note that a string value differs from a numeric value by quoting "" or ''. For instance, type the command **date = '31/12/2016'** and **type(date)**. Jupyter recognises this variable as a string, written in short form as **str**.

```
In [7]: date = '31/12/2016'
        type(date)

Out[7]: str
```

Reassigning variables

It should be noticed that each variable can only store one value at the same time. However, a variable can be easily reassigned different values. Let us enter the following command. As you can see, the output changes from 7 to 3 while the variable name remains same.

```
In [8]: a = 7
        a = 3
        print(a)

3
```

Name rules

It is useful to know that Python has strict rules for naming variables. Firstly, Python does not allow leaving spaces within a variable name. In other words, Python only allows an one-word name. If you want to create a variable like **my value = 1**, Jupyter will return an error message. To avoid such mistake, it is recommended to type either **myvalue = 1** or **my_value = 1**. Note that an underscore is the only special character which can be used to name variables in Python.

Secondly, a variable cannot begin with a number. Try to type the command **3value = 1** and see whether Jupyter can correctly execute this line.

Thirdly, it should be noticed that Python's variable names are case-sensitive. **value = 1** and **Value = 2** will be two different variables.

Finally, do not use any term which is a keyword set by Python such as **return**, **in**, etc. These will lead Jupyter to mix up the actual purposes of the code. To check all Python keywords, you can type the following command (see In [14])

```
In [9]: my value = 1


          File "<iPython-input-9-286a8b996a2d>", line 1
        my value = 1
               ^
    SyntaxError: invalid syntax



In [10]: myvalue = 1
```

```
In [11]: my_vlaue = 1

In [12]: 3value = 1


          File "<iPython-input-12-0b1e89f2a049>", line 1
        3value = 1
             ^
    SyntaxError: invalid syntax



In [13]: value = 1
         Value = 2
         print(value)
         print(Value)

1
2


In [14]: import keyword
         keyword.kwlist

Out[14]: ['False',
          'None',
          'True',
          'and',
          'as',
          'assert',
          'break',
          'class',
          'continue',
          'def',
          'del',
          'elif',
          'else',
          'except',
          'finally',
          'for',
          'from',
          'global',
          'if',
          'import',
          'in',
          'is',
          'lambda',
          'nonlocal',
          'not',
          'or',
```

```
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

2.2 Whitespace

Apart from the rules of naming variables, Python also has strict grammatical rules, like any other language. It is important to indent some particular lines of code, thus it can be structured in an appropriate format and correctly recognised. An example of using whitespace is shown in the following, where there is an indentation before the command **return**. This can be done by hitting the button **TAB**. Hitting **TAB** once will lead Jupyter create one whitespace. If you want to close the whitespace, press **SHIFT** and **TAB**. Sometimes, Jupyter will automatically indent the cursor if you hit **ENTER** to change a line when necessary. However, we recommend users to check whitespaces with caution as Jupyter will not run the code if it is badly formatted.

```
In [15]: def func():
             return print('hello world')

         func()
```

```
hello world
```

```
In [16]: def func():
         return print('hello world')

         func()
```

```
  File "<iPython-input-16-68ea9780d691>", line 2
    return print('hello world')
        ^
IndentationError: expected an indented block
```

2.3 Comments

It is a good habit for programmers to add comments to necessary lines of code for better readability. Sometimes, a few words of explanation will make the programme easier understand. In most cases, comments are helpful when you look back at your code or you are working on it in a team where others need to collaborate. In Python, a comment is incorporated by typing any explanations preceded by a # sign, and thus Jupyter will ignore this line of text and will not run it as code. Alternatively, you can use the keyboard shortcuts **CONTROL** and **/**. Moreover, in a Jupyter NoteBook, there are a few ways to make comments. Apart from using # to comment out lines of code, you can write a few sentences in a empty cell above or below and convert it to markdown.[8]

[8] A further benefit of Jupyter NoteBooks is that they support plain text writing and can easily generate documentation.

2.4 Mathematical operations

Let us now become familiarised with simple mathematical operations in Python. The basic operations such as addition, subtraction, multiplication and division can be easily implemented by typing **+**, **-**, *, /. For example, we can practice these commands by typing the following code.

```
In [17]: add_two_num = 1 + 2
         subtract_two_num = 10 - 2
         multiply_two_num = 17*2
         divide_two_num = 24/6

         print(add_two_num)
         print(subtract_two_num)
         print(multiply_two_num)
         print(divide_two_num)
```

```
3
8
34
4.0
```

There is a slight difference when calculating exponents (i.e., the power of numbers). For exponent operations, we use the sign $**$ instead of the $*$ operator. To implement more advanced mathematical operations such as square roots, inner products, etc., you will need the assistance of Python's built-in libraries such as **NumPy** or user-defined functions, which will be explored later in this guide.

Doing maths with user-defined functions

It is natural for finance researchers to deal with more complex data calculations in the real world. Examples range from a simple case of transforming price data to continuous logarithmic returns, to a complex project of calculating the Dickey–Fuller test critical values by simulation. Now, let us take a look at how to implement maths operations by a user-defined function. Supposed that a variable needs to be squared (i.e., taking a power of two for the value); this can be easily done by defining a function.[9] Firstly, we create a data series by entering the command **my_data = 12**. Then a custom function is typed as shown in the following cell. After that, the function needs to be called outside if we implement it, and the results will be displayed in the out cell.

```
In [18]: my_data = 12
         def func(x):
             y = x**2
             return y

         func(my_data)
```

```
Out[18]: 144
```

[9] Alternatively, you can directly calculate the square of a value by using the $**$ operator. Here, the example is made for the purpose of familiarising you with how to implement a user-defined function.

2.5 Two libraries: Pandas and NumPy

Let us now look at two built-in Python libraries. When it comes to financial data management, two of the most frequently used Python packages/libraries are **Pandas** and **NumPy**. Pandas, which is the abbreviation for Panel Data Analysis, is a popular choice to handle datasets. It provides an array of functions to cover different requirements such as reading and aggregating data, etc. The other library, **NumPy**, has a large collection of high-level mathematical functions, and therefore is particularly suited for data calculation. The combination of applying **Pandas** and **NumPy** can easily handle most data management tasks. Given the scope of this guide, we cannot fully cover the libraries' contents by introducing every function, but instead we are going to cover several of the most important functions which will be employed to serve as examples. If readers are interested in the other applications of these two libraries, some tutorials are recommended: http://Pandas.pydata.org/Pandas-docs/stable/.

A function can be easily called only after you import the library. This is why the command for importing libraries is always put at the beginning of a script. To employ the **Pandas** library, we type **import Pandas as pd** in the NoteBook. This line means that Jupyter is told to import the Pandas library and assigns it a short name, pd, for ease of reference. Similarly, we can also import the library **NumPy** and refer to it with the abbreviation np by the command **import NumPy as np**.

```
In [1]: import Pandas as pd
        import NumPy as np
```

2.6 Data input and saving

One of the first steps of every statistical analysis is importing the dataset to be analysed into the software. Depending on the format of the data there are different ways of accomplishing this task. Since we mainly handle data using the Pandas library, its reading function will be elaborated upon. **Pandas** provides a data import function **read_excel** if the dataset you want to read is stored in an Excel workfile. Sometimes the dataset might be in a csv format, in which case the corresponding function **read_csv** should be employed.

Now let us import a dataset to see how the steps would be performed. The dataset that we want to import into Jupyter is the Excel file 'fund_managers.xlsx'. Since we have imported the Pandas library, the file can be read into the NoteBook by typing the following code and then hitting **SHIFT** and **ENTER** leads Jupyter to execute the line.

```
In [2]: data = pd.read_excel('C:/Users/tao24/Desktop/fund_managers.xlsx')
        type(data)

Out[2]: Pandas.core.frame.DataFrame
```

Note that the imported dataset is stored in a variable called **data**. However, this variable does not belong to any of data types as stated in previous sections if you print its type. This is a new object called a DataFrame, which is designed to store data such as an Excel sheet. In the Pandas documentation, a DataFrame is described as a structure with at least two columns. On the other hand, it is sometimes the case that you might come across a similar data structure called a **Series**. The difference between these two containers is minimal, in that a **Series** is a one-dimensional labelled array whereas a **DataFrame** is a two-dimensional array. In most cases, the data set that finance researchers use is a two-or-more-dimensional array. Therefore, it is important to become familiar with how to implement some basic applications based on it.

2.7 Data description and calculation

Reading: Brooks (2019, section 2.3

Summary statistics

Once you have imported the data, you may want to obtain an idea of their features and characteristics, and you would probably want to check that all the values have been correctly imported and that all variables are stored in the correct format.

There are several functions to examine and describe the data. It is often useful to visually inspect the data. This can be done by the command **print(data)**. Alternatively, DataFrame contains a built-in function called **head()**. You can use it directly through the command **data.head()**. It allows you to access the first five rows of data by default settings.

```
In [3]: print(data)
```

```
    Year  Risky Ricky  Safe Steve  Ricky Ranks  Steve Ranks
0   2005           24           9            2            3
1   2006           18           7            3            4
2   2007            4           5            7            6
3   2008          -23          -8           13           13
4   2009          -12          -3           11           12
5   2010            1           3            8            8
6   2011            7           4            6            7
7   2012           12           2            5           10
8   2013           -6           3            9            8
9   2014          -14           6           12            5
10  2015           -7           2           10           10
11  2016           56          19            1            1
12  2017           14          12            4            2
```

```
In [4]: data.head()
```

```
Out[4]:    Year  Risky Ricky  Safe Steve  Ricky Ranks  Steve Ranks
        0  2005           24           9            2            3
        1  2006           18           7            3            4
        2  2007            4           5            7            6
        3  2008          -23          -8           13           13
        4  2009          -12          -3           11           12
```

Let us have a look at how to calculate summary statistics given the 'fund_managers.xlsx' file. This Excel workfile contains the annual data on the performance of two fund managers who are working for the same company. By calculating the mean, standard deviation, skewness and kurtosis, we can compare the performance of the two fund managers. Supposed we would like to compute the summary statistics for Ricky, the first step we need to do is picking up the corresponding column from the DataFrame **data**. This can be done by typing the command **data['Risky Ricky']**. Next, continue to type the function **mean** followed by the selected column if you want to obtain the average value of the series.[10] Likewise, typing the command as follows, all the corresponding results will be displayed.

[10] As Pandas is a rich data analysis library, basic statistical operations such as mean, standard deviation, skewness and kurtosis have been embedded.

```
In [5]: print(data['Risky Ricky'].mean(), data['Safe Steve'].mean())
        print(data['Risky Ricky'].std(), data['Safe Steve'].std())
        print(data['Risky Ricky'].skew(), data['Safe Steve'].skew())
        print(data['Risky Ricky'].kurtosis(), data['Safe Steve'].kurtosis())

5.6923076923076925 4.6923076923076925
20.36084729484759 6.612924408366485
1.11545231491 0.294506522695
2.10421426375 1.45903450796
```

If you want to get a idea of the general information content of a DataFrame, such as the mean, standard deviation, minimum and maximum values, Pandas provides a function **describe**. Type the command **data.describe()** to see the results.

```
In [6]: data.describe()
```

Out[6]:

	Year	Risky Ricky	Safe Steve	Ricky Ranks	Steve Ranks
count	13.00000	13.000000	13.000000	13.00000	13.000000
mean	2011.00000	5.692308	4.692308	7.00000	6.846154
std	3.89444	20.360847	6.612924	3.89444	3.782551
min	2005.00000	-23.000000	-8.000000	1.00000	1.000000
25%	2008.00000	-7.000000	2.000000	4.00000	4.000000
50%	2011.00000	4.000000	4.000000	7.00000	7.000000
75%	2014.00000	14.000000	7.000000	10.00000	10.000000
max	2017.00000	56.000000	19.000000	13.00000	13.000000

Now let us interpret the results. If we look at the mean returns, we can see that Ricky's is a full percentage point higher (5.69 versus 4.69). But when comparing the variation of performance over time, it is clear that his returns are more volatile. If we look at the higher moment values for Ricky and Steve, the skewness figures are, respectively, 1.12 and 0.29, which favours Ricky. On the other hand, the kurtosis values are 2.10 and 1.46, respectively, and therefore Steve's is better because his distribution is more concentrated around the centre.

Additionally, we might be interested to know the relationship between Ricky's performance and Steve's. We can get an idea of this by computing the correlation between the two sets of returns. In Python, this can be done by using the Pandas function **corr**, which will compute pairwise correlation of columns. The default setting calculates a Pearson correlation between these two fund managers of 0.87, indicating that the two series do move very closely together over time.

```
In [7]: data.corr()
```

Out[7]:

	Year	Risky Ricky	Safe Steve	Ricky Ranks	Steve Ranks
Year	1.000000	0.142928	0.385059	0.021978	-0.197996
Risky Ricky	0.142928	1.000000	0.870048	-0.934285	-0.766741
Safe Steve	0.385059	0.870048	1.000000	-0.796004	-0.944866
Ricky Ranks	0.021978	-0.934285	-0.796004	1.000000	0.763700
Steve Ranks	-0.197996	-0.766741	-0.944866	0.763700	1.000000

Internal rate of return

So far, we have been focusing on the functions embedded in Pandas library, but it is also worth exploring the NumPy package where a large number of advanced mathematical operating functions are provided. For example, we sometimes know both the present value of a particular set of cashflows and all of the future cashflows but we wish to calculate the discount rate. It is very straightforward to determine this, which is also known as the internal rate of return (IRR), in Python.

Suppose that we have a series of cashflows. This can be set up in Python by creating a Pandas series. Let us enter the command in the following to store these entries. The parameter **index** is set up for a series as an index column, which is years in this case. The second parameter we specify is the series' name. Since there is only one column for the series object, we leave a string value 'Cashflow' for this argument. Finally, the value of series needs to be completed by adding the **data** argument.

```
In [8]: cashflow = pd.Series(index=[0,1,2,3,4,5], name='Cashflow',\
                             data=[-107,5,5,5,5,105])
        print(cashflow)

        np.irr(cashflow)
```

```
0   -107
1      5
2      5
3      5
4      5
5    105
Name: Cashflow, dtype: int64


Out[8]: 0.034517484085994976
```

Then, in another new line, we simply write the command **np.irr(cashflow)**. Then hitting **SHIFT** and **ENTER** leads Jupyter to calculate the IRR, which is 0.0345 in this case.

2.8 An example: calculating summary statistics for house prices

Reading: Brooks (2019, sections 2.3 and 2.4

By now, you will have observed a broad picture of how to implement some basic applications in Python. In this section, we will apply these functions/commands again, refreshing this knowledge by a new example.

Suppose, for example, the dataset that we want to import into the NoteBook is contained in the Excel file **UKHP.xls**. First, we import the two libraries **Pandas** and **NumPy**, and we read the file using the function **read_excel**. Here, we input one additional parameter when reading the data apart from the file path. Specifically, we set up the first column of the imported dataset **Month** as an index column instead of Pandas default index, which is 0, 1, 2 ... . It is often the case that we are handling data in Pandas when a file contains time-series data, or, in other words, a series of data points with corresponding date time values. This makes things easier when we later want to slice and index the time-series data, as the index points and column names are perfect coordinates to locate any values inside a DataFrame.

```
In [1]: import Pandas as pd
        import NumPy as np

        data = pd.read_excel('C:/Users/tao24/Desktop/UKHP.xls', index_col=0)
        data.head()
```

```
Out[1]:             Average House Price
        Month
        1991-01-01         53051.721106
        1991-02-01         53496.798746
        1991-03-01         52892.861606
        1991-04-01         53677.435270
        1991-05-01         54385.726747
```

We want to have a look at the structure of the imported data and check whether all variables are stored in the correct format. This can be done by either using the command **print(data)** or **data.head()**. As can be seen in Out [1], the data are correctly formatted. Moreover, additional information can be accessed by applying some mathematical functions – for example, by typing the command **data.mean()**, **data.std()**,**data.skew()**, **data.kurtosis()** to calculate the mean, standard deviation, skewness and kurtosis of the sample data, respectively. Alternatively, the command **data.describe()** also gives several summary statistics such as the mean, standard deviation, minimum and maximum values.

```
In [2]: print(data['Average House Price'].mean())
        print(data['Average House Price'].std())
        print(data['Average House Price'].skew())
        print(data['Average House Price'].kurtosis())

        data.describe()
```

```
124660.48446512519
56387.16566469951
-0.11014547215
-1.59192678501
```

```
Out[2]:        Average House Price
        count           327.000000
        mean         124660.484465
        std           56387.165665
        min           49601.664241
        25%           61654.141609
        50%          150946.108249
        75%          169239.278727
        max          211755.925562
```

Often, you need to change the data by creating new variables, changing the content of existing data series, or by converting the data type. In the following, we will focus on some of the most important Python features to manipulate and reassign the data, although this is not exhaustive.

Suppose that we want to calculate the log changes in house prices. The first step achieving this is to find a mathematical operation/function. Since there is no existing function in Python to achieve this task, we need to break down the calculation and design a function in Jupyter. To do so, we can type the following command and create a use-defined function called **LogDiff**. With respect to the contents of this function, we first create a one-period lagged house prices series by taking the function **shift()** from the Pandas library. Then the log function is called from the NumPy library by the command **np.log()**. Next, we take the original series and divide it by the lagged series inside the logarithmic function. Once finished, the results need to be outputted by the command **return**. Note that it is important to check the format of the code, including that a colon has been added at the of end of the function name and that the whitespace has been correctly set up.

```
In [3]: def LogDiff(x):
            x_diff = 100*np.log(x/x.shift(1))
            return x_diff
```

After the user-defined function has been set up, we then call this function to generate a new variable **dhp** by typing the following command. You can see the data now have a new series, with the first observation missing due to the calculation. The command **data.describe()** illustrates this even more clearly as there are 326 observations for the 'dph' variable vs that of 'Average House Price' being 327. Furthermore, a new DataFrame/Series can be created if we want to dump old variables and store new variables for further analysis. To avoid potentially undesirable errors, any missing values can be dropped by the Pandas function **dropna**.

```
In [4]: data['dhp'] = LogDiff(data['Average House Price'])
        data.head()
```

```
Out[4]:             Average House Price       dhp
        Month
        1991-01-01         53051.721106       NaN
        1991-02-01         53496.798746  0.835451
        1991-03-01         52892.861606 -1.135343
        1991-04-01         53677.435270  1.472432
        1991-05-01         54385.726747  1.310903
```

```
In [5]: data.describe()
```

```
Out[5]:        Average House Price         dhp
        count           327.000000  326.000000
        mean         124660.484465    0.424402
        std           56387.165665    1.114586
        min           49601.664241   -3.464027
        25%           61654.141609   -0.256025
        50%          150946.108249    0.447332
        75%          169239.278727    1.145618
        max          211755.925562    3.731686
```

```
In [6]: data1 = pd.DataFrame({'dhp':LogDiff(data['Average House Price'])})
        data1 = data1.dropna()
        data1.head()
```

```
Out[6]:                  dhp
        Month
        1991-02-01  0.835451
        1991-03-01 -1.135343
        1991-04-01  1.472432
        1991-05-01  1.310903
        1991-06-01  1.318181
```

2.9 Plots

Python supports a wide range of graph types that can be found under the **matplotlib.pyplot** library, including line graphs, bar graphs, pie charts, mixed-line graphs, scatterplots and others. A variety of parameters/arguments permits the user to select the line type, colour, border characteristics, headings, shading and scalding, including dual scale graph, etc. Legends can be created if desired, and output graphs can be incorporated into other applications using copy-and-paste, or by exporting in another file format, including Windows metafiles, portable network graphics, or pdf. For the latter, you simply right click and select the **Save image as** option in the 'Graph' window and select the desired file format.

Assume that we would like to plot a line plot of the 'Average House Price' series. To do so, we firstly import the library **matplotlib.pyplot**. In the following, we create a graph object by the command **plt.figure(1, figsize=(20,10))**. The parameters of the image size inside the parentheses can be adjusted if desired, or left at the default settings by the command **plt.figure(1)**. Next, we select the **plot** function and specify the plotted line and its label. Additionally, there are a number of other options available using further functions, e.g., plotting the grid line on the background or formatting the Y and X axes, the titles and the legend. Finally, type **plt.show()** and output the graph in Figure 14.

```
In [7]: import matplotlib.pyplot as plt

        plt.figure(1)
        plt.plot(data['Average House Price'], label='hp')

        plt.xlabel('Date')
        plt.ylabel('Average House Price')
        plt.title('Graph')
        plt.grid(True)

        plt.legend()
        plt.show()
```

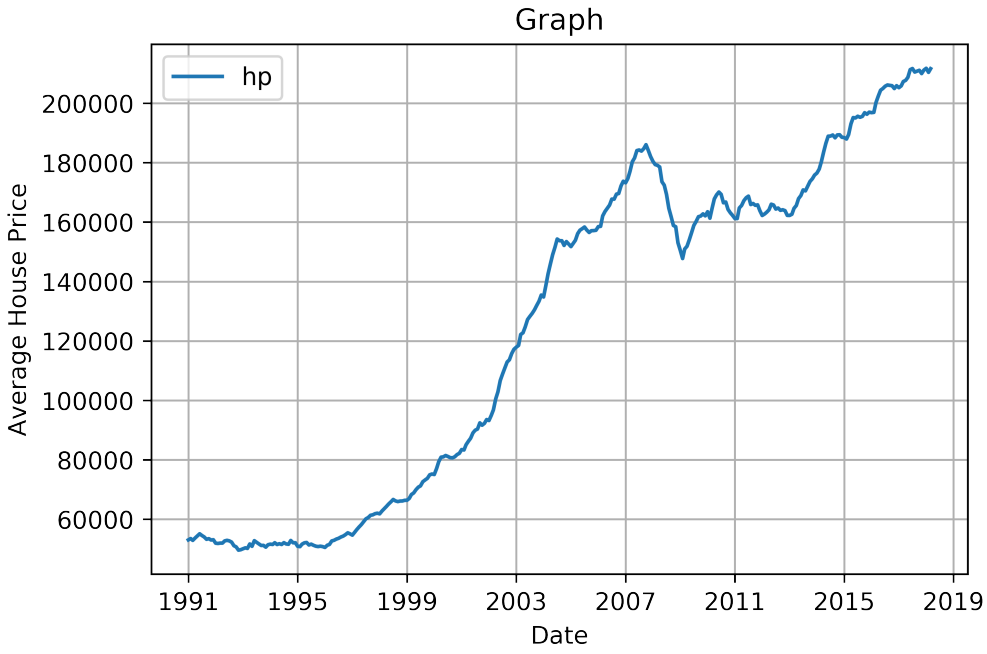


Figure 14: Line plot of the Average House Price Series

Another commonly used plot is a histogram, which provides a graphical illustration of the distribution of a series. It is basically a bar chart that demonstrates the proportion of the series that falls within a specific range of values. To show how to create a histogram using Python we will employ the 'dhp' series. Again, we firstly create a graph object. The function **hist** from the library is employed, with plotted data, bins, edge colours and line's width being set up. Type **plt.show()** and the histogram of the 'dhp' series should resemble that in the Out [8] and Figure 15.

```
In [8]: plt.figure(2)
        plt.hist(data1['dhp'], 20, edgecolor='black', linewidth=1.2)
        plt.xlabel('dhp')
        plt.ylabel('Density')
        plt.title('Histogram')
        plt.show()
```

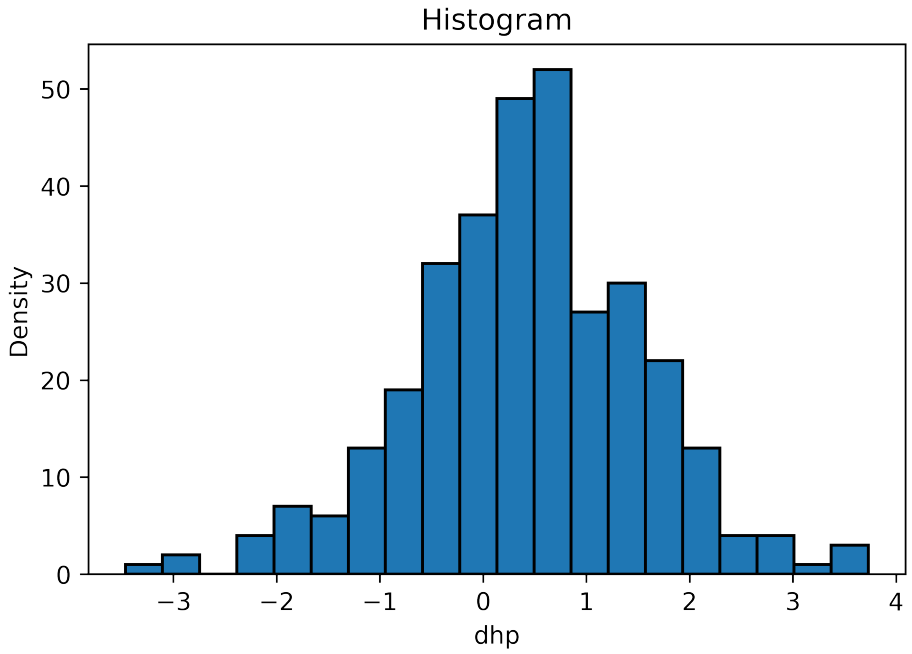


Figure 15: Histogram of the Average House Price Series

2.10 Saving data and results

Data generated in Python can be exported to other applications, e.g., Microsoft Excel. To export data, call the Pandas function **to_excel** to save the data in an Excel workfile or save it as a csv file by another function, **to_csv**. In the function's brackets, you can then specify a file name and location for the new file.

You can export single outputs by just selecting and copying the results, then right-clicking and choosing the 'Copy' option required. For example, you can copy (and paste the results into Excel).

```
In [9]: data.to_excel('C:/Users/tao24/Desktop/UKHP_workfile.xls')
```

Additionally, there are several other commands that can be used to save the data in a DataFrame or Series format. One useful command can be applied by importing the library **pickle**. For example, creating a pickle file 'UKHP' and storing the finalised series 'Average House Price' and 'dhp' in the DataFrame. To do so, we first specify the location of the new file, and then type the following command. By using the function **dump**, we save the data onto a local drive.

To retrieve the object we saved by pickle, simply use the function **load** and assigning the object to a new variable. Note that there are several modes when saving and re-loading a workfile in Python, it can be observed that we use the 'wb' and 'rb' mode to dump and read the file, respectively.

```
In [10]: import pickle

         with open('C:/Users/tao24/Desktop/UKHP.pickle', 'wb') as handle:
             pickle.dump(data, handle)
```

```
In [11]: with open('C:/Users/tao24/Desktop/UKHP.pickle', 'rb') as handle:
             data = pickle.load(handle)
```

Finally, it is important for users to record both the results and the list of commands in a Jupyter NoteBook in order to be able to later reproduce the work and remember what has been done after some time. It is good practice to keep the NoteBook so that you can easily replicate the work. We remind readers to save the NoteBook frequently.

3 Simple linear regression - estimation of an optimal hedge ratio

Reading: Brooks (2019, section 3.3)

This section shows how to run a bivariate regression using Python. 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.[11]

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, we need to import this Excel workfile into a Python Jupyter NoteBook. For this, we type the code as follows (see In [1]). The first four lines are to import the necessary Python built-in packages and the next two lines are to read data from the Excel workfile into Jupyter. Note that we import one new library **statsmodels** to perform the econometric tests. We will very often use this library in later sections of this guide so we will leave a more detailed description until then. Since the data file contains monthly datetime observations, we specify that column as the time-series index of the Pandas DataFrame. In order to have a look at the data and verify some data entries, just type variable name **data.head()** and then hit **SHIFT** plus **ENTER**. By doing so, the first five rows of the imported DataFrame can be displayed in the following cell. As Out [2] shows, the data stored in the NoteBook has three different columns, Date, Spot and Futures, respectively.

```
In [1]: import Pandas as pd
        import NumPy as np
        import statsmodels.formula.api as smf

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        data = pd.read_excel(abspath + 'SandPhedge.xls', index_col=0)

        data.head()
```

```
Out[1]:                    Spot  Futures
        Date
        1997-09-01  947.280029   954.50
        1997-10-01  914.619995   924.00
        1997-11-01  955.400024   955.00
        1997-12-01  970.429993   979.25
        1998-01-01  980.280029   987.75
```

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 first define the regression formula as **Spot** $\sim$ **Futures**. Python then recognises that these two variable names come from the DataFrame columns. **Spot** is specified as the dependent variable while **Futures** is the independent variable. Thus, we are trying to explain the prices of the spot asset with the prices of the corresponding futures asset.

The built-in package Statsmodels automatically includes a constant term in the linear regression, and therefore you do not need to specifically include it among the independent variables. If you do not wish to include a constant term in the regression, you can type **Spot** $\sim$ **Futures - 1**. The added **fit()** function followed by **smf.ols(formula, data)** estimates an OLS regression based on the whole

[11] See also chapter 9 of Brooks (2019).

sample. This function allows you to customise the regression specification, e.g., by employing a different treatment of the standard errors. However, for now we stick with the default specification and type **print(results.summary())** in order to obtain regression results. The results will be the output immediately below In [3].

```
In [2]: formula = 'Spot ~ Futures'
        results = smf.ols(formula, data).fit()
        print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   Spot   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
Method:                 Least Squares   F-statistic:                 1.005e+06
Date:                Mon, 30 Jul 2018   Prob (F-statistic):               0.00
Time:                        14:53:33   Log-Likelihood:                -826.86
No. Observations:                 247   AIC:                             1658.
Df Residuals:                     245   BIC:                             1665.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -2.8378      1.489     -1.906      0.058      -5.771       0.095
Futures        1.0016      0.001   1002.331      0.000       1.000       1.004
==============================================================================
Omnibus:                      245.415   Durbin-Watson:                   1.326
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            10091.682
Skew:                          -3.814   Prob(JB):                         0.00
Kurtosis:                      33.371   Cond. No.                     5.05e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 5.05e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

The parameter estimates for the intercept ($\hat{\alpha}$) and slope ($\hat{\beta}$) are -2.8378 and 1.0016, respectively. A large number of other statistics are also presented in the regression output -- the purpose and interpretation of these will be discussed later.

We can now proceed to estimate the regression based on first differences. That is to say, we run the analysis based on returns of the S&P500 index instead of price levels; so the next step is to transform the 'Spot' and 'Futures' 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 generate a new data series of continuously compounded returns, we define a new function in Python to achieve this calculation (see In [4]). Specifically, we create a user-defined function

called **LogDiff** where the input parameter is a Pandas column.[12] To calculate the log difference, we first obtain a column which lags one period. This can be done by typing the command **x.shift(1)**. Then, we take the log transformation of the difference between the original and lagged series. The function log comes from the NumPy library, so we can use this function by entering the command **np.log(x/x.shift(1))**. Next, we want to output the result expressed as a percentage. Therefore, the column is scaled by multiplying it by 100. It is worth noting that the first price observation will be lost when return series is computed. However, the first data point will still be displayed in the DataFrame as nan since Python keeps the length of the DataFrame intact. To avoid this pitfall, we employ the Pandas **dropna()** function remove it. Finally, we return the newly-calculated column.

We have to call the function itself outside the function content. This can be done by typing the command **LogDiff(data['Spot'])**. Meanwhile, we also rebuild the **data** DataFrame which previously stored the prices data. A newly created DataFrame can be defined by using the Pandas **DataFrame** function. In the bracket of the **DataFrame** function, we specify two column names: **ret_spot** and **ret_future** respectively. To fill out their values, the **LogDiff** function is called to compute the new series. Once finished, we can repeat the process as stated above to print the finalised DataFrame. As can be seen in Out [4], the newly created DataFrame starts from October 1997 instead of September 1997 and the column names have changed from **Spot** and **Futures**to **ret_spot** and **ret_future**.

```
In [3]: def LogDiff(x):
            x_diff = 100*np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data = pd.DataFrame({'ret_spot' : LogDiff(data['Spot']),
                             'ret_future':LogDiff(data['Futures'])})
        data.head()
```

```
Out[3]:             ret_future  ret_spot
        Date
        1997-10-01   -3.247557 -3.508608
        1997-11-01    3.299927  4.362145
        1997-12-01    2.507563  1.560914
        1998-01-01    0.864266  1.009901
        1998-02-01    6.159189  6.807837
```

Before proceeding to estimate the regression, we can examine a number of descriptive statistics together and measures of association between the series. For the summary statistics, we type **data.describe()** since Pandas contains a summary statistics function called **describe()** (see In [5]).

We observe that the two return series are quite similar as based on their mean values, and standard deviations, as well as their minimum and maximum values, as one would expect from economic theory. Note that the number of observations has reduced from 247 for the levels of the series to 246 when we computed the returns (as one observation is 'lost' in constructing the $t-1$ value of the prices in the returns formula).

```
In [4]: data.describe()
```

```
Out[4]:         ret_future    ret_spot
        count  246.000000  246.000000
```

[12] In Python, the name of the input variable in an user-defined function does not need to be the same as the name outside the function. For example, the input variable name is **x** whereas the actual input is **data['Spot']**.

```
mean      0.414017    0.416776
std       4.419049    4.333323
min     -18.944697  -18.563647
25%      -1.931400   -1.831388
50%       0.997641    0.918522
75%       3.133588    3.276468
max      10.387184   10.230659
```

Now we can estimate a regression for the returns. We can follow the steps described above and specify 'ret_spot' as the dependent variable and 'ret_future' as the independent variable. As shown in In [5], the revised formula statement is **'ret_spot ~ ret_future'**, and we repeat the regression procedure again. The result of testing the returns data is displayed in the following area.

```
In [5]: formula = 'ret_spot ~ ret_future'
        results = smf.ols(formula, data).fit()
        print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                ret_spot   R-squared:                       0.989
Model:                            OLS   Adj. R-squared:                  0.989
Method:                 Least Squares   F-statistic:                 2.147e+04
Date:                Mon, 30 Jul 2018   Prob (F-statistic):          7.54e-240
Time:                        14:53:33   Log-Likelihood:                -157.16
No. Observations:                 246   AIC:                             318.3
Df Residuals:                     244   BIC:                             325.3
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.0131      0.029      0.444      0.658      -0.045       0.071
ret_future     0.9751      0.007    146.543      0.000       0.962       0.988
==============================================================================
Omnibus:                       48.818   Durbin-Watson:                   2.969
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              671.062
Skew:                          -0.016   Prob(JB):                    1.91e-146
Kurtosis:                      11.091   Cond. No.                         4.45
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

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).

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 do not have the results ready on the Python main screen, reload the 'SandPhedge.xls' file now and re-estimate the returns regression using the steps described in the previous section. While we have so far mainly focused on the coefficient estimates, i.e., the α and β estimates, the built-in package Statsmodels has also calculated 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 the 't' in the column heading. They are the statistics for tests 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 ('Intercept') in the first row of numbers and $H_0 : \beta = 0$ versus $H_1 : \beta \neq 0$ for 'ret_future' in the second row.

Let us focus on the t-ratio for the α estimate first. We see that with a value of only 0.444, 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 'ret_future', the t-ratio is high with 146.543 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 a 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 the **summary** function of Statsmodels 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 Python do this work. For this we use the Statsmodels function 'f_test'.

We assume you have conducted the procedure as stated in the previous section, that is to import the Excel workfile into Python. Similarly, we first specify the regression formula where **Futures** is set as explanatory variable and **Spot** is the dependent variable. Note that Statsmodels automatically includes the constant term in the formula statement. Next, we type the regression hypotheses **'Futures = 1'**. This is because we want to test a linear hypothesis that the coefficient estimate for 'Futures = 1'.

```
In [1]: import Pandas as pd
        import NumPy as np
        import statsmodels.formula.api as smf

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD' \
                        '/QMF Book/book Ran/data files new/Book4e_data/'
        data = pd.read_excel(abspath + 'SandPhedge.xls', index_col=0)
```

```
In [2]: formula = 'Spot ~ Futures'
        hypotheses = 'Futures = 1'
        results = smf.ols(formula, data).fit()
        f_test = results.f_test(hypotheses)
        print(f_test)
```

```
<F test: F=array([[ 2.58464834]]), p=0.10919227950361698, df_denom=245, df_num=1>
```

The output line 'F test' reports all the statistics. First we find the test statistics: 'F=array([[2.58464834]])', which states that the value of the F-test is around 2.58. The corresponding p-value is 0.11, stated in the next position. As it is larger than 0.10, we clearly cannot reject the null hypothesis that the coefficient estimate is equal to 1. The last two numbers present the total number of observations and the degrees of freedom for this test respectively.

We can also perform hypothesis testing on the first difference regressions. For this we need to calculate the log returns of the series by the function in In [3]. After that, we want to test the null hypothesis that $H_0 : \beta = 1$ on the coefficient estimate for 'ret_future', so we can just type the formula **'ret_spot** $\sim$ **ret_future'** and hypothesis testing command **'ret_future = 1'** to generate the corresponding F-statistics for this test.

```
In [3]: def LogDiff(x):
            x_diff = 100*np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data = pd.DataFrame({'ret_spot' : LogDiff(data['Spot']),
                             'ret_future':LogDiff(data['Futures'])})
```

```
In [4]: formula = 'ret_spot ~ ret_future'
        hypotheses = 'ret_future = 1'

        results = smf.ols(formula, data).fit()
        f_test = results.f_test(hypotheses)
        print(f_test)
```

```
<F test: F=array([[ 14.02981315]]), p=0.00022456591761704512, df_denom=244, df_num=1>
```

With an F-statistic of 14.03 and a corresponding p-value of nearly zero, we find that the null hypothesis can be rejected at the 1% significance level.

5 Estimation and 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 need to import this data file into Python. As in the previous example, we first need to import several necessary built-in Python libraries such as Pandas, NumPy, Statsmodels and matplotlib. Sometimes, these libraries have very long names so we often give them a shorter one in order to easily re-write later in the script. For example, we often short the library Statsmodels.formula.api as smf and matplotlib.pyplot as plt. Next, we specify the directory where the Excel file 'capm.xls' is found. To do so, type the following line of code (see in [1]). Then the Pandas **read_excel** function is called, with the filepath and several arguments input in the brackets. As for the previous section, we add the arguments **index_col=0** in order to create a time-series index column for the DataFrame.

The imported data file contains monthly stock prices for the S&P500 index ('SandP'), the four companies: Ford ('FORD'), General Motors ('GE'), Microsoft ('MICROSOFT') and Oracle ('ORACLE'), as well as the 3-month US Treasury bill rates ('USTB3M') from January 2002 until February 2018. You can check that Python has imported the data correctly by printing the variables in the command area.

```
In [1]: import Pandas as pd
        import NumPy as np
        import statsmodels.formula.api as smf
        import matplotlib.pyplot as plt

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD' \
                        '/QMF Book/book Ran/data files new/Book4e_data/'
        data = pd.read_excel(abspath + 'capm.xls', index_col=0)

        data.head()
```

Out[1]:

Date	SandP	FORD	GE	MICROSOFT	ORACLE	USTB3M
2002-01-01	1130.199951	15.30	37.150002	31.855000	17.260000	1.68
2002-02-01	1106.729980	14.88	38.500000	29.170000	16.620001	1.76
2002-03-01	1147.390015	16.49	37.400002	30.155001	12.800000	1.83
2002-04-01	1076.920044	16.00	31.549999	26.129999	10.040000	1.75
2002-05-01	1067.140015	17.65	31.139999	25.455000	7.920000	1.76

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 15 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, we can simply copy the function that is designed to compute log returns from the previous section. Recall that the function **shift(1)** is used to instruct Python to create one-period lagged observations of the series.

By calling the user-defined function **LogDiff**, Python creates a new data series that will contain continuously compounded returns on the S&P500. Next, we need to repeat these steps for the stock prices of the four companies. To accomplish this, we can create a new DataFrame with newly-created series computed by the same process as described for the S&P500 index. We rename the column

names in the Pandas DataFrame function. To fill out the DataFrame values, the customised function is called every time when we specify a new column. For example, we input **data['SandP']** into the function **LogDiff** followed by a new column **'ret_sandp'**.

Regarding **'ersandp'** and **'erford'**, we need to deduct the risk free rate, in our case the 3-month US Treasury bill rate, from the continuously compounded returns in order to transform the returns into excess 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. To do so we simply type **data['USTB3M']/12**. Now that the risk free rate is a monthly variable, we can compute excess returns. For example, to generate the excess returns for the S&P500, we type **LogDiff(data['SandP']) - data['USTB3M']/12**. We similarly generate excess returns for the four stock returns. Finally, you can check the data series by printing the DataFrame (see In [2]).

```
In [2]: def LogDiff(x):
            x_diff = 100*np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data = pd.DataFrame({'ret_sandp' : LogDiff(data['SandP']),
                             'ret_ford' : LogDiff(data['FORD']),
                             'USTB3M' : data['USTB3M']/12,
                             'ersandp' : LogDiff(data['SandP']) - data['USTB3M']/12,
                             'erford' : LogDiff(data['FORD']) - data['USTB3M']/12})
        data.head()
```

Out[2]:

Date	USTB3M	erford	ersandp	ret_ford	ret_sandp
2002-01-01	0.140000	NaN	NaN	NaN	NaN
2002-02-01	0.146667	-2.930147	-2.245153	-2.783480	-2.098486
2002-03-01	0.152500	10.121111	3.455511	10.273611	3.608011
2002-04-01	0.145833	-3.162375	-6.484299	-3.016541	-6.338466
2002-05-01	0.146667	9.668039	-1.058964	9.814706	-0.912297

Before running the CAPM regression, we can plot the data series to examine whether they appear to move together. We do this for the S&P500 and the Ford series. We choose **matplotlib.pyplot** in the Python Library. We first define a new figure object and leave its size as the default setting in Jupyter. This can be implemented by **plt.figure(1)**. However, you can easily adjust the figure by adding parameters inside the parentheses (e.g., **plt.figure(1, figsize=(20,10))**). The number 1 refers to the serial number of the figure object that you are going to plot in Python.

Next, we can plot the two series. The function **plot** from the library can achieve this purpose. Note that each **plot** function can only take one data series. Therefore, if you wish to plot multiple lines in one image, you have to call the function once again. Meanwhile, the **label** argument is recommend when we plot multiple series together because it will clearly show the name of each line on the figure. Additionally, the **matplotlib.pyplot** needs to be manually set up for optional parameters including the x-axis, y-axis, title, etc. For example, if we want the figure to be plotted with grid lines, we can achieve this by typing the command **plt.grid(True)**. Moreover, Python will not automatically display the label of the plotted series unless you specify the command **plt.legend()**. Finally, we end up with the command **plt.show()** and the figure can be seen below the In [3] Figure 16.

```
In [3]: plt.figure(1)
        plt.plot(data['ersandp'], label='ersandp')
        plt.plot(data['erford'], label='erford')

        plt.xlabel('Date')
        plt.ylabel('ersandp/erford')
        plt.title('Graph')
        plt.grid(True)

        plt.legend()
        plt.show()
```

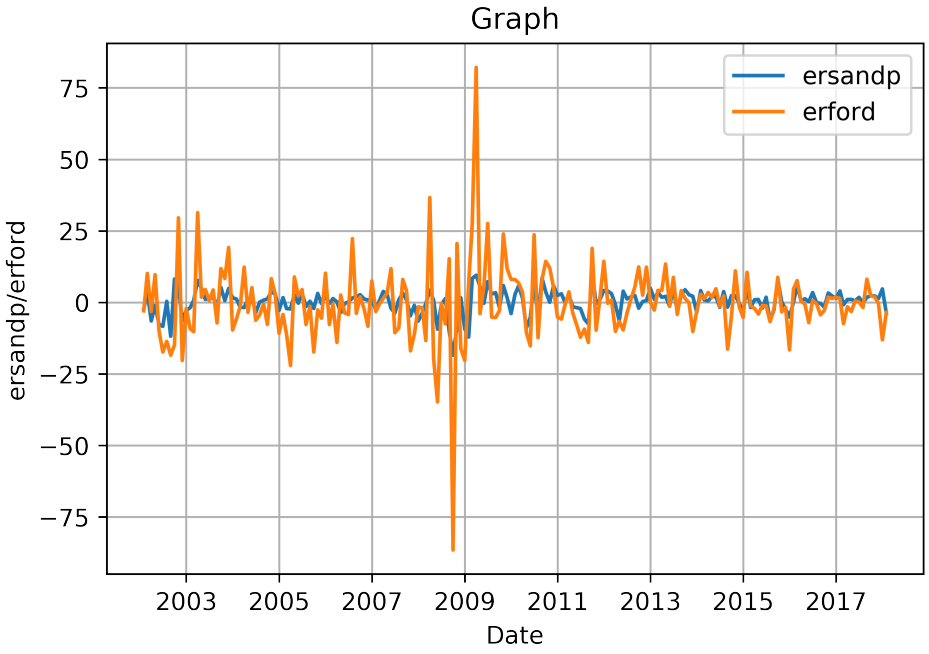


Figure 16: Time-Series Plot of Two Series

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, we first call the function **plt.scatter()**. This time, we can input two series **data['ersandp']** and **data['erford']** into the parentheses because a scatter plot needs pairs of data points. Following the same command, we can now see the scatter plot of the excess S&P500 return and the excess Ford return as depicted in the following output cell in Figure 17.

```
In [4]: plt.figure(2)

        plt.scatter(data['ersandp'], data['erford'])

        plt.xlabel('ersandp')
```

```
plt.ylabel('erford')
plt.title('Graph')
plt.grid(True)

plt.show()
```

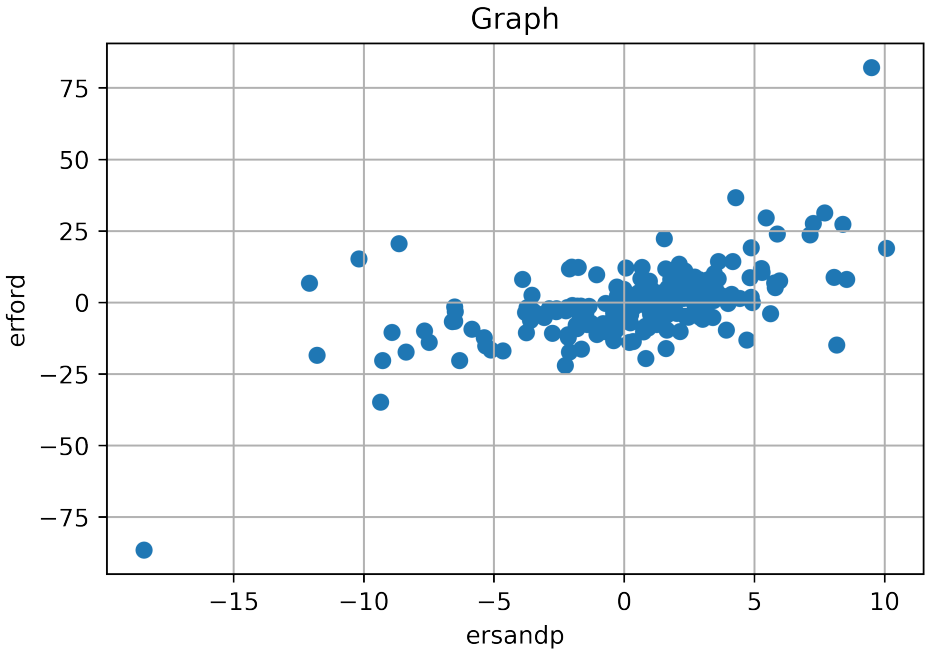


Figure 17: Scatter Plot of two Series

To estimate the CAPM equation, we employ the **statsmodels.formula.api**. As usual, we first run the regression formula which is **'erford** $\sim$ **ersandp'**. 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'.[13] Next, we use the **smf.ols()** function followed by **fit()**. Note that we keep the default setting this in case although the function **fit()** can allow for different treatments of the standard errors. Finally, the result is outputted by **summary()**. The OLS regression results appear in the following code cell (see In [5]).

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

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

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) estimate is 1.8898 with a t-ratio of 9.862 and a corresponding p-value of 0.000. This suggests

[13] Remember that the Statsmodels command **formula()** automatically includes a constant in the regression; thus, we do not need to manually include it among the independent variables.

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 –0.9560 with a t-ratio of –1.205 and a p-value of 0.230. Thus, we cannot reject the null hypothesis that the α estimate is different from 0, indicating that Ford stock does not seem to significantly out-perform or under-perform the overall market.

```
In [5]: formula = 'erford ~ ersandp'
        results = smf.ols(formula, data).fit()
        print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 erford   R-squared:                       0.337
Model:                            OLS   Adj. R-squared:                  0.334
Method:                 Least Squares   F-statistic:                     97.26
Date:                Mon, 30 Jul 2018   Prob (F-statistic):           8.36e-19
Time:                        15:11:43   Log-Likelihood:                -735.26
No. Observations:                 193   AIC:                             1475.
Df Residuals:                     191   BIC:                             1481.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -0.9560      0.793     -1.205      0.230      -2.520       0.608
ersandp        1.8898      0.192      9.862      0.000       1.512       2.268
==============================================================================
Omnibus:                       71.412   Durbin-Watson:                   2.518
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              677.387
Skew:                           1.079   Prob(JB):                    8.08e-148
Kurtosis:                      11.921   Cond. No.                         4.16
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Assume that we want to test the null hypothesis of whether the value of the population coefficient on 'ersandp' is equal to 1. How can we achieve this? The answer is to use the **statsmodels.formula.api** library to launch a **f_test** and then to specify **hypotheses = 'ersandp = 1'**. By printing the result of the f_test, the statistic for this hypothesis test appears in the output window. The F-statistic of 21.56 with a corresponding p-value of 0 (at least up to the fourth decimal place) implies that the null hypothesis of the CAPM beta of Ford stock being 1 is convincingly rejected and hence the estimated beta of 2.026 is significantly different from 1.[14]

[14]This is hardly surprising given the distance between 1 and 2.026. 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.

```
In [6]: # F-test: hypothesis testing
        formula = 'erford ~ ersandp'
        hypotheses = 'ersandp = 1'

        results = smf.ols(formula, data).fit()
        f_test = results.f_test(hypotheses)
        print(f_test)
```

```
<F test: F=array([[ 21.5604412]]), p=6.3653210360356986e-06, df_denom=191, df_num=1>
```

6 Sample output for multiple hypothesis tests

Reading: Brooks (2019, section 4.4)

This example uses the 'capm.pickle' workfile constructed in the previous section. So in case you are starting a new session, re-load the Python workfile and re-estimate the CAPM regression equation for the Ford stock. Let us first review how to save and re-load a Python workfile. To do so, we first import the **pickle** library. We then need to specify the path where the workfile is going to be saved. Next, a new pickle file called **camp.pickle** is created. The function **pickle.dump** saves the data into the newly created workfile. The 'wb' mode inside the bracket is the one of file modes that Python reads or writes.[15] To check whether you have saved the workfile properly, we recommend you look up the folder where the file is saved. Alternatively, you can import this workfile by the function**pickle.load** and print to check the data. As can be seen in In [1], the difference between the saved and re-loaded file is minimal. We only need to change one command function and file mode.

```
In [1]: import Pandas as pd
        import NumPy as np
        import pickle

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD' \
                            '/QMF Book/book Ran/data files new/Book4e_data/'
        data = pd.read_excel(abspath + 'capm.xls', index_col=0)

        def LogDiff(x):
            x_diff = 100*np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data = pd.DataFrame({'ret_sandp' : LogDiff(data['SandP']),
                            'ret_ford' : LogDiff(data['FORD']),
                            'USTB3M' : data['USTB3M']/12,
                            'ersandp' : LogDiff(data['SandP']) - data['USTB3M']/12,
                            'erford' : LogDiff(data['FORD']) - data['USTB3M']/12})

        with open(abspath + 'capm.pickle', 'wb') as handle:
            pickle.dump(data, handle)

In [2]: with open(abspath + 'capm.pickle', 'rb') as handle:
            data = pickle.load(handle)
```

Now suppose 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. First, we import the Statsmodel library by typing **import Statsmodels.formula.api as smf**. To construct the regression specification, we write the formula statement **'erford ~ ersandp'** and the hypotheses statement **'ersandp = Intercept = 1'**. Regression results are calculated by the two consecutive functions **smf.ols(formula, data).fit()**. Note that the first function **ols** takes the formula and data while the second function **fit()** takes the standard error robustness methods. Here, we leave it as the default setting. To obtain the specific *F*-test statistic, we use the function **f_test** to take the hypothesis statement. Finally, the output can be printed in the following cell.

[15] The 'wb' indicates that the file is opened for writing in binary mode. There are also modes such as 'rb', 'r+b', etc. Here we are not going to explain each of these modes in detail.

```
In [3]: import statsmodels.formula.api as smf
        # F-test: multiple hypothesis tests
        formula = 'erford ~ ersandp'
        hypotheses = 'ersandp = Intercept = 1'

        results = smf.ols(formula, data).fit()
        f_test = results.f_test(hypotheses)
        print(f_test)
```

```
<F test: F=array([[ 12.9437402]]), p=5.3492561526069516e-06, df_denom=191, df_num=2>
```

In the *Output* window (see In [3]), Python 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, '(1) ersandp = 1' and '(2) _Intercept = 1'. Looking at the value of the *F*-statistic of 12.94 with a corresponding *p*-value of roughly 0, we conclude that the null hypothesis, $H_0 : \beta_1 = 1$ and $\beta_2 = 1$, is strongly rejected at the 1% significance level.

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 previous sections to estimate multiple regressions in Python. 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 13 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 the following maturities: three months, six months, one year, three years, five years 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 several libraries and the dataset 'macro.xls' into Python. The index column of the imported DataFrame is set by the argument **index_col=0**.[16]

```
In [1]: import Pandas as pd
        import NumPy as np
        import statsmodels.formula.api as smf

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        data = pd.read_excel(abspath + 'macro.xls', index_col=0)
        data.head()
```

```
Out[1]:             MICROSOFT       SANDP    CPI   INDPRO  M1SUPPLY   CCREDIT  \
        Date
        1986-03-01   0.095486  238.899994  108.8  56.5414     624.3  606.7990
        1986-04-01   0.111979  235.520004  108.6  56.5654     647.0  614.3669
        1986-05-01   0.121528  247.350006  108.9  56.6850     645.7  621.9152
        1986-06-01   0.106771  250.839996  109.5  56.4959     662.8  627.8910
        1986-07-01   0.098958  236.119995  109.5  56.8096     673.4  633.6083

                    BMINUSA  USTB3M  USTB10Y
        Date
        1986-03-01     1.50    6.76     7.78
        1986-04-01     1.40    6.24     7.30
        1986-05-01     1.20    6.33     7.71
        1986-06-01     1.21    6.40     7.80
        1986-07-01     1.28    6.00     7.30
```

Now that we have prepared the dataset we can start with the actual analysis. The first stage is to generate a set of changes or differences for each of the variables, since 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

[16] Recall that we need to set the first column of 'macro.xls' as an index column, thus the command will be **index_col=0**.

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).[17]

To transform the variables, we re-construct the DataFrame where each variable is defined and computed as follows: To compute changes or differences, we set up a user-defined function to facilitate the calculation as used in previous sections.

```
In [2]: def LogDiff(x):
            x_diff = 100*np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data = pd.DataFrame({'dspread' : data['BMINUSA'] - \
                                         data['BMINUSA'].shift(1),
                             'dcredit' : data['CCREDIT'] - \
                                         data['CCREDIT'].shift(1),
                             'dprod' : data['INDPRO'] - \
                                       data['INDPRO'].shift(1),
                             'rmsoft' : LogDiff(data['MICROSOFT']),
                             'rsandp' : LogDiff(data['SANDP']),
                             'dmoney' : data['M1SUPPLY'] - \
                                        data['M1SUPPLY'].shift(1),
                             'inflation' : LogDiff(data['CPI']),
                             'term' : data['USTB10Y'] - data['USTB3M'],
                             'dinflation' : LogDiff(data['CPI']) - \
                                            LogDiff(data['CPI']).shift(1),
                             'mustb3m' : data['USTB3M']/12,
                             'rterm' : (data['USTB10Y'] - data['USTB3M']) - \
                                       (data['USTB10Y'] - data['USTB3M']).shift(1),
                             'ermsoft' : LogDiff(data['MICROSOFT']) - \
                                         data['USTB3M']/12,
                             'ersandp' : LogDiff(data['SANDP']) - \
                                         data['USTB3M']/12})
        data.head()
```

Out[2]:

Date	dcredit	dinflation	dmoney	dprod	dspread	ermsoft	ersandp
1986-03-01	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1986-04-01	7.5679	NaN	22.7	0.0240	-0.10	15.413171	-1.944918
1986-05-01	7.5483	0.459855	-1.3	0.1196	-0.20	7.655834	4.373351
1986-06-01	5.9758	0.273590	17.1	-0.1891	0.01	-13.479167	0.867757
1986-07-01	5.7173	-0.549452	10.6	0.3137	0.07	-8.099084	-6.547514

[17] 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.

```
            inflation   mustb3m     rmsoft    rsandp  rterm  term
Date
1986-03-01        NaN  0.563333        NaN       NaN    NaN  1.02
1986-04-01  -0.183993  0.520000  15.933171 -1.424918   0.04  1.06
1986-05-01   0.275862  0.527500   8.183334  4.900851   0.32  1.38
1986-06-01   0.549452  0.533333 -12.945833  1.401091   0.02  1.40
1986-07-01   0.000000  0.500000  -7.599084 -6.047514  -0.10  1.30
```

We save the data for for future research since the calculation of these series is handy. Recall that although there are several ways of keeping data file, we employ the pickle library and save all columns.

```
In [3]: import pickle

        with open(abspath + 'macro.pickle', 'wb') as handle:
            pickle.dump(data, handle)
```

We can now run the regression. To create a regression specification, we first write the formula statement **'ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney + dspread + rterm'**. The formula string will automatically add a constant term, unless the user types '-1' at the end of the formula. Next, the regression function **ols** is called from **statsmodels.formula.api**, written in shorthand as smf, with both the formula and data input. The function **fit** following it allows for different treatments of the regression standard errors. Here we leave it as the default setting. Finally, typing the command **print(results.summary())** obtains all the information from the regression.

```
In [4]: formula = 'ermsoft ~ ersandp + dprod + dcredit + \
                           dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit()
        print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                ermsoft   R-squared:                       0.345
Model:                            OLS   Adj. R-squared:                  0.333
Method:                 Least Squares   F-statistic:                     28.24
Date:                Thu, 09 Aug 2018   Prob (F-statistic):           3.52e-31
Time:                        11:22:44   Log-Likelihood:                -1328.3
No. Observations:                 383   AIC:                             2673.
Df Residuals:                     375   BIC:                             2704.
Df Model:                           7
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      1.3260      0.475      2.789      0.006       0.391       2.261
ersandp        1.2808      0.094     13.574      0.000       1.095       1.466
dprod         -0.3030      0.737     -0.411      0.681      -1.752       1.146
dcredit       -0.0254      0.027     -0.934      0.351      -0.079       0.028
dinflation     2.1947      1.264      1.736      0.083      -0.291       4.681
dmoney        -0.0069      0.016     -0.441      0.659      -0.037       0.024
```

```
dspread        2.2601      4.140      0.546      0.585      -5.881      10.401
rterm          4.7331      1.716      2.758      0.006       1.359       8.107
==============================================================================
Omnibus:                       21.147   Durbin-Watson:                   2.097
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               63.505
Skew:                          -0.006   Prob(JB):                     1.62e-14
Kurtosis:                       4.995   Cond. No.                         293.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Let us take a few minutes to examine the main regression results. Which of the variables has a statistically significant impact on the Microsoft excess returns? Readers can use their knowledge of the effects of the financial and macroeconomic environment on stock returns to examine whether the coefficients have their expected signs and whether the sizes of the parameters are plausible.

The regression F-statistic takes a value of 28.24 (third row, top right corner). Remember that this tests the null hypothesis of all the slope parameters being jointly zero. The p-value of zero attached to the test statistic shows that this null hypothesis should be rejected. However, a number of parameter estimates are not significantly different from zero -- specifically, those on the 'dprod', 'dcredit', 'dmoney' and 'dspread' variables.

```
In [5]: hypotheses = 'dprod = dcredit = dmoney = dspread = 0'

        f_test = results.f_test(hypotheses)
        print(f_test)

<F test: F=array([[ 0.41387856]]), p=0.7986453783444192, df_denom=375, df_num=4>
```

Let us test the null hypothesis that the parameters on these four variables are jointly zero using an F-test. To achieve this, we write the hypotheses using the string literals **'dprod = dcredit = dmoney = dspread = 0'** following the In [4] cell. Applying the function **f_test** from the instance **result** and typing the command **f_test = results.f_test(hypotheses)**.[18] As shown in In [5], the resulting F-test statistic value is 0.414 with p-value 0.799; there are four restrictions and 375 usable observations. This suggests that the null hypothesis cannot be rejected.

[18] An instance is a type of variable in object-oriented programming such as Python.

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 selects first the variable with the lowest p-value (largest t-ratio) if it were included, then the variable with the second lowest p-value conditional upon the first variable already being included, and so on. The procedure continues until the next lowest p-value relative to those already included variables is larger than some specified threshold value, then the selection stops, with no more variables being incorporated into the model.

We want to conduct a stepwise regression which will automatically select the most important variables for explaining the variations in Microsoft stock returns. Since there is no module available for this specific application, we design a user-defined function to fit this purpose, the details of which are listed in the following cell.

```
In [1]: import statsmodels.formula.api as smf
        import Pandas as pd
        import NumPy as np

        def forward_selected(data, endog, exg):
            '''
            Linear model designed by forward selection based on p-values.

            Parameters:
            -----------
            data : Pandas DataFrame with dependent and independent variables

            endog: string literals, dependent variable from the data

            exg: string literals, independent variable from the data

            Returns:
            --------
            res : an "optimal" fitted Statsmodels linear model instance
                  with an intercept selected by forward selection
            '''
            remaining = set(data.columns)
            remaining = [e for e in remaining if (e not in endog)&(e not in exg)]
            exg = [exg]

            scores_with_candidates = []
            for candidate in remaining:
                formula = '{} ~ {}'.format(endog,' + '.join(exg + [candidate]))

                score = smf.ols(formula, data).fit().pvalues[2]
                scores_with_candidates.append((score, candidate))
            scores_with_candidates.sort()

            for pval,candidate in scores_with_candidates:
                if pval < 0.2:
                    exg.append(candidate)
```

```
        formula = '{} ~ {}'.format(endog, ' + '.join(exg))
        res = smf.ols(formula, data).fit()
        return res
```

Let us break down this function line by line. As usual, we import the Pandas, NumPy and Statsmodels libraries for the convenience of calling functions. Next, name a function 'forward_selected' with several of the necessary inputs added in parentheses. The details of the inputs and outputs are written between three single quote marks immediately below. 'data' contains all of the variables that might be taken in the regression specification. 'endog' is the endogenous variable of the regression whereas the main exogenous variable is specified as 'exg'. To obtain all of the variables' names except 'endog' and 'exg' from 'data', we type the commands **remaining = set(data.columns)** and **remaining = [e for e in remaining if (e not in endog)&(e not in exg)]**. Specifically, the former line takes all of the columns' names into a set and the latter creates a list of variables conditional upon the variable not being included in 'endog' or 'exg'.[19] The next line converts the string literals 'exg' to a list.

To select variables with the potentially lowest p-value, we create a new list object named 'scores_with_candidates'. By using a **for** loop, we take each variable from the list **remaining** and fill out the formula statement. Specifically, there are three functions combined under the loop. The first function **formula = ' '** produces a formula statement where two curly brackets take the dependent and independent variables accordingly. The second function **format** following is used to fill out the two empty brackets, i.e., the 'endog' and the 'exg's, respectively. Furthermore, it can be observed that there are two inputs inside the brackets where the first is the 'endog' and the second is an argument. The **' + '.join(exg)** converts a list of string values into single string literals separated by '+' signs. The final aim of this command is to produce a formula, like 'ermsoft ~ ersandp + dinflation' for example.

The next combined functions command **score = smf.ols(formula, data).fit().pvalues[2]** examines the candidate variable's p-value under each regression specification, with **ols**, **fit** and **pvalues** being called, respectively.[20] It is natural to record all candidates with their p-values after performing the regression. This can be done by calling the list class built-in function **append** where the pairwise combination of p-value and variable name is stored in each loop. Finally, we sort the list in ascending order based on p-values.

We filter the candidates with p-values lower than 0.2 (this threshold can be adjusted if required). Specifically, we create a new for loop to go through each p-value and variable, then set up an if conditional statement where only the variables that meet the condition can then be added to the 'exg' list. Once finished, we perform the regression analysis again with all of the selected variables and return the regression instance.

Let us implement stepwise regressions using the function Import library **pickle** and re-load the dataset 'macro' which was saved in the previous section. We assign the data to the variable 'data' and print the first five rows by the command **data.head()**.

```
In [2]: import pickle

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'macro.pickle', 'rb') as handle:
```

[19] The second command is somewhat complex. This is basically an iterator which enables Jupyter to create a container combined with a conditional if statement. The finalised variable 'remaining' will be a list object.

[20] The function **pvalues** returns the p-value. The command **pvalues[2]** specifies the candidate's p-value since 0 and 1 refer to the 'Intercept' and 'exg', respectively.

```
        data = pickle.load(handle)

    data = data.dropna() # drop the missing values for some columns
    data.head()
```

Out[2]:

Date	ermsoft	ersandp	dprod	dcredit	dinflation	dmoney	dspread	rterm
1986-06-01	-13.479167	1.330990	-0.1891	4.2358	0.273590	17.1	0.01	0.02
1986-07-01	-8.099084	-6.479720	0.3137	5.0952	-0.549452	10.6	0.07	-0.10
1986-08-01	-0.474167	6.403095	-0.0748	3.7534	0.182482	5.0	0.18	0.18
1986-09-01	-1.326843	-9.376901	0.1135	7.1188	0.272271	6.1	-0.15	0.62
1986-10-01	31.160969	4.880987	0.2696	7.7641	-0.364050	7.7	0.07	0.01

To perform the forward selection procedure, the defined function **forward_selected** is called with three parameters input, where 'ermsoft' is specified as the dependent variable and 'ersandp' is the initial independent variable. The returned instance will be saved in the variable **res**, which contains a number of details. For example, we can type the command **print(res.model.formula)** and see the final regression specification. As can be seen below in In [3], the regression formula is 'ermsoft $\sim$ ersandp + rterm + dinflation', suggesting that the excess market return, the term structure, and unexpected inflation variables have been included while the money supply, default spread and credit variables have been omitted.

If you want to access the full details of the regression table, simply type the command **print(res.summary())**. You can then check whether the variable with corresponding p-value (t-ratio) is indeed lower than the pre-specified threshold. There are also other values displayed in the table. For example, the adjusted R-squared (second row, top right corner) is 0.334, indicating that selected independent variables can explain roughly 33% of the variation in the dependent variable 'ermsoft'. Alternatively, this figure can be accessed by the command **print(res.rsquared_adj)**.

```
In [3]: res = forward_selected(data,'ermsoft','ersandp')

        print(res.model.formula)
```

```
ermsoft ~ ersandp + rterm + dinflation
```

```
In [4]: print(res.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                ermsoft   R-squared:                       0.339
Model:                            OLS   Adj. R-squared:                  0.334
Method:                 Least Squares   F-statistic:                     64.58
```

```
Date:                Tue, 31 Jul 2018   Prob (F-statistic):           1.02e-33
Time:                        11:34:44   Log-Likelihood:                -1323.8
No. Observations:                 381   AIC:                             2656.
Df Residuals:                     377   BIC:                             2671.
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      1.0225      0.404      2.533      0.012       0.229       1.816
ersandp        1.2595      0.092     13.642      0.000       1.078       1.441
rterm          4.8402      1.721      2.812      0.005       1.456       8.224
dinflation     2.2094      1.217      1.816      0.070      -0.183       4.602
==============================================================================
Omnibus:                       21.219   Durbin-Watson:                   2.075
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               64.060
Skew:                           0.009   Prob(JB):                     1.23e-14
Kurtosis:                       5.009   Cond. No.                         18.7
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

```
In [5]: print(res.rsquared_adj)

0.33421129163
```

Note that a stepwise regression can be executed in different ways – e.g., either 'forward' or 'backward'. Here, our example is analysed only by the 'forward' method. In the interests of brevity, we leave employing other approaches as an exercise for interested readers.

8 Quantile regression

Reading: Brooks (2019, section 4.10)

To illustrate how to run quantile regressions using Python, we will now employ the simple CAPM beta estimation conducted in the previous section. We re-open the 'capm.pickle' workfile using the **pickle** module and assign it to the variable **data**. We select the function **quantreg**, then write 'erford' as the dependent variable and 'ersandp' as the independent variable and input the dataset. As usual, in the module **statsmodels.formula.api**, we do not need to specify the constant as the function will automatically include a constant term. Next, we type the function **fit** and input the argument **q=0.5**. This tells Python the quantile regression we want to implement is 0.5, which is an Ordinary-Least-Squares (OLS) regression.

```
In [1]: import Pandas as pd
        import NumPy as np
        import statsmodels.formula.api as smf
        import pickle
        import matplotlib.pyplot as plt

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                          'QMF Book/book Ran/data files new/Book4e_data/'

        with open(abspath + 'capm.pickle', 'rb') as handle:
            data = pickle.load(handle)
        data = data.dropna()
```

```
In [2]: # regression
        # quantile(50)
        res = smf.quantreg('erford ~ ersandp', data).fit(q=0.5)
        print(res.summary())
```

```
                         QuantReg Regression Results
==============================================================================
Dep. Variable:                 erford   Pseudo R-squared:               0.1724
Model:                       QuantReg   Bandwidth:                       5.340
Method:                 Least Squares   Sparsity:                        16.78
Date:                Thu, 23 Aug 2018   No. Observations:                  193
Time:                        20:55:13   Df Residuals:                      191
                                        Df Model:                            1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -1.4896      0.606     -2.457      0.015      -2.685      -0.294
ersandp        1.4384      0.146      9.822      0.000       1.150       1.727
==============================================================================
```

While this command (see In [2]) 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. To print several quantile estimations, we can apply the **for** loop. To do so, we firstly need to create an array starting from 0.1 to 1.0 with an interval of 0.1. This can be done by selecting the

arrange function from **NumPy** (see In [3]). We then set up a loop and iterate each value from the array object we created. Under each loop, the command shown in In [2] can be repeated. Meanwhile we only need to revise the argument of the function **fit(q=0.5)** to **fit(q=x)**. For example, when x takes the value 0.3, q will then be assigned this value, which tells Python to produce the 0.3 quantile regression.

For each quantile (**0.1 to 0.9**), Python reports two estimates together with their respective test statistics: the β-coefficient on 'ersandp' and the coefficient for the constant term.

```
In [3]: # Simultaneous-quantile regression
        # 10 20 30 40 50 60 70 80 90, ten quantiles
        quantiles = np.arange(0.10, 1.00, 0.10)

        for x in quantiles:
            print('-------------------------------------------------')
            print('{0:0.01f} quantile'.format(x))
            res = smf.quantreg('erford ~ ersandp', data).fit(q=x)
            print(res.summary())
```

```
-----------------------------------------------
0.1 quantile
                         QuantReg Regression Results
==============================================================================
Dep. Variable:                 erford   Pseudo R-squared:               0.2036
Model:                       QuantReg   Bandwidth:                       5.609
Method:                  Least Squares   Sparsity:                        47.54
Date:                 Thu, 23 Aug 2018   No. Observations:                  193
Time:                         20:55:13   Df Residuals:                      191
                                         Df Model:                            1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept    -11.9291      1.045    -11.420      0.000     -13.989      -9.869
ersandp        2.3404      0.348      6.720      0.000       1.653       3.027
==============================================================================
-----------------------------------------------
0.2 quantile
                         QuantReg Regression Results
==============================================================================
Dep. Variable:                 erford   Pseudo R-squared:               0.2005
Model:                       QuantReg   Bandwidth:                       4.710
Method:                  Least Squares   Sparsity:                        25.26
Date:                 Thu, 23 Aug 2018   No. Observations:                  193
Time:                         20:55:13   Df Residuals:                      191
                                         Df Model:                            1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -7.3496      0.733    -10.021      0.000      -8.796      -5.903
ersandp        1.8044      0.186      9.694      0.000       1.437       2.172
```

```
==============================================================================
------------------------------------------------
0.3 quantile
                         QuantReg Regression Results
==============================================================================
Dep. Variable:                 erford   Pseudo R-squared:                0.1923
Model:                        QuantReg   Bandwidth:                       5.074
Method:                  Least Squares   Sparsity:                        19.85
Date:                 Thu, 23 Aug 2018   No. Observations:                  193
Time:                         20:55:13   Df Residuals:                      191
                                         Df Model:                            1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -4.8783      0.661     -7.381      0.000      -6.182      -3.575
ersandp        1.6596      0.163     10.187      0.000       1.338       1.981
==============================================================================
------------------------------------------------
0.4 quantile
                         QuantReg Regression Results
==============================================================================
Dep. Variable:                 erford   Pseudo R-squared:                0.1753
Model:                        QuantReg   Bandwidth:                       5.192
Method:                  Least Squares   Sparsity:                        18.09
Date:                 Thu, 23 Aug 2018   No. Observations:                  193
Time:                         20:55:13   Df Residuals:                      191
                                         Df Model:                            1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -3.3479      0.641     -5.223      0.000      -4.612      -2.084
ersandp        1.5005      0.155      9.669      0.000       1.194       1.807
==============================================================================
------------------------------------------------
0.5 quantile
                         QuantReg Regression Results
==============================================================================
Dep. Variable:                 erford   Pseudo R-squared:                0.1724
Model:                        QuantReg   Bandwidth:                       5.340
Method:                  Least Squares   Sparsity:                        16.78
Date:                 Thu, 23 Aug 2018   No. Observations:                  193
Time:                         20:55:13   Df Residuals:                      191
                                         Df Model:                            1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -1.4896      0.606     -2.457      0.015      -2.685      -0.294
ersandp        1.4384      0.146      9.822      0.000       1.150       1.727
==============================================================================
```

```
------------------------------------------------
0.6 quantile
                         QuantReg Regression Results
==============================================================================
Dep. Variable:                  erford   Pseudo R-squared:               0.1679
Model:                        QuantReg   Bandwidth:                      5.155
Method:                  Least Squares   Sparsity:                       17.05
Date:                 Thu, 23 Aug 2018   No. Observations:                 193
Time:                         20:55:13   Df Residuals:                     191
                                         Df Model:                           1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -0.0188      0.605     -0.031      0.975      -1.212       1.174
ersandp        1.2967      0.150      8.658      0.000       1.001       1.592
==============================================================================
------------------------------------------------
0.7 quantile
                         QuantReg Regression Results
==============================================================================
Dep. Variable:                  erford   Pseudo R-squared:               0.1611
Model:                        QuantReg   Bandwidth:                      4.881
Method:                  Least Squares   Sparsity:                       20.29
Date:                 Thu, 23 Aug 2018   No. Observations:                 193
Time:                         20:55:13   Df Residuals:                     191
                                         Df Model:                           1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      1.7568      0.676      2.597      0.010       0.423       3.091
ersandp        1.5283      0.182      8.396      0.000       1.169       1.887
==============================================================================
------------------------------------------------
0.8 quantile
                         QuantReg Regression Results
==============================================================================
Dep. Variable:                  erford   Pseudo R-squared:               0.1357
Model:                        QuantReg   Bandwidth:                      4.653
Method:                  Least Squares   Sparsity:                       29.42
Date:                 Thu, 23 Aug 2018   No. Observations:                 193
Time:                         20:55:13   Df Residuals:                     191
                                         Df Model:                           1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      4.0013      0.856      4.674      0.000       2.313       5.690
ersandp        1.6199      0.251      6.454      0.000       1.125       2.115
==============================================================================
------------------------------------------------
```

```
0.9 quantile
                         QuantReg Regression Results
==============================================================================
Dep. Variable:                 erford   Pseudo R-squared:               0.1063
Model:                       QuantReg   Bandwidth:                      4.810
Method:                 Least Squares   Sparsity:                       88.43
Date:                Thu, 23 Aug 2018   No. Observations:                 193
Time:                        20:55:13   Df Residuals:                     191
                                        Df Model:                           1
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     10.4563      1.951      5.359      0.000       6.608      14.305
ersandp        1.8473      0.699      2.644      0.009       0.469       3.225
==============================================================================
```

It would be unwise to print all the summary tables from the regressions across quantiles due to the volume of results. To present all of the key parameters of each model specification in an efficient way, a user-defined function can be written. To do so, we create a function called **model_paras** with two inputs: **data** and **quantiles**. We list the details of the input parameters and output results at the beginning to help users quickly understand the aim of the function. There are two purposes of this function. First, it will produce all of the necessary statistics including an α-coefficient for the constant term, a β-coefficient on 'ersandp', and lower and upper bounds of 95 % confidence intervals. Second, it will output all the fitted value of regression specifications.

Let us now examine this function. First, we need to create one empty list object and one empty dictionary to save the results accordingly. Following the same procedure for getting all the quantile summary tables, we repeat the command **res = smf.quantreg('erford $\sim$ ersandp', data).fit(q=q)** under the **for** loop. To further obtain the key parameters instead of printing whole summary tables, we obtain the former by typing the following commands: **alpha = res.params['Intercept']**, **beta = res.params['ersandp']**, **lb_pval = res.conf_int().loc['ersandp'][0]** and **ub_pval = res.conf_int().loc['ersandp'][1]**.[21]

Additionally, we want to obtain the fitted value of y across quantiles. This can be easily achieved by the built-in method **fittedvalues** from the quantile regression instance **res**. Once finished, the fitted regression values are required to be stored into the corresponding dictionary cell. For example, when the first quantile regression has been performed (i.e., q is 0.1), the dictionary **y_pred** records its key as 0.1 while the corresponding value is assigned by the 0.1 quantile regression predicted series. Next, q takes the value of 0.2, which is the second quantile regression. The dictionary **y_pred[0.2]** stores the fitted value of 0.2 quantile regression with the key 0.2, and so on.

Likewise, it is also necessary to record all regression statistics (α, β and bounds of confidence intervals) under each specification. With a slightly different implementation, we call the built-in function **append** followed by each list under the loop. This function can update the list, as the name suggests. Thus, the row in the list increases with each iteration. Once the command is finished, we exit the loop by closing the whitespace. Finally, to better present the results, we still need to convert these temporary variables into a Pandas DataFrame before returning the final results. Specifically, we simply create a new DataFrame **quantreg_res** with names corresponding to each column for the list **parameters**. For the dictionary, **y_pred**, we convert it to the DataFrame in the same way but

[21] The detail of these functions can be seen on the following web site https://www.Statsmodels.org/stable/generated/Statsmodels.regression.linear_model.RegressionResults.html

without additional arguments. The difference between the two commands arises from the fact that **parameters** is a list without column names while those of **y_pred** are saved with keys (i.e., the latter have column names).

```
In [4]: def model_paras(data, quantiles):
            '''
            Parameters:
            -----------
                data: Pandas DataFrame with dependent and independent variables
                quantiles: quantile number

            Returns:
            -----------
                quantreg_res: Pandas DataFrame with model parameters for each
                              quantile regression specification
                y_hat: Pandas DataFrame with all the fitted value of y
            '''
            parameters = []
            y_pred = {}
            for q in quantiles:
                res = smf.quantreg('erford ~ ersandp', data).fit(q=q)
                # obtain regression's parameters
                alpha = res.params['Intercept']
                beta = res.params['ersandp']
                lb_pval = res.conf_int().loc['ersandp'][0]
                ub_pval = res.conf_int().loc['ersandp'][1]
                # obtain the fitted value of y
                y_pred[q] = res.fittedvalues
                # save results to lists
                parameters.append((q,alpha,beta,lb_pval,ub_pval))

            quantreg_res = pd.DataFrame(parameters, columns=['q', 'alpha', \
                                                             'beta','lb','ub'])
            y_hat = pd.DataFrame(y_pred)
            return quantreg_res, y_hat
```

We can implement this function to obtain a more concise table with all the key parameters (see In[5]). If you wish to obtain other regression results, you can change the function accordingly, such as including t-ratio or p-value, for example.

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.

```
In [5]: quantreg_paras, y_hats = model_paras(data, quantiles)

In [6]: print(quantreg_paras)

     q      alpha      beta        lb        ub
0  0.1 -11.929063  2.340422  1.653477  3.027366
1  0.2  -7.349600  1.804414  1.437253  2.171575
2  0.3  -4.878332  1.659638  1.338293  1.980982
3  0.4  -3.347896  1.500533  1.194442  1.806624
4  0.5  -1.489629  1.438449  1.149566  1.727332
5  0.6  -0.018844  1.296706  1.001296  1.592115
6  0.7   1.756783  1.528341  1.169294  1.887389
7  0.8   4.001336  1.619863  1.124772  2.114954
8  0.9  10.456266  1.847333  0.469290  3.225376


In [9]: y_hats.head()

Out[9]:                  0.1        0.2        0.3        0.4        0.5       0.6  \
        Date
        2002-02-01 -17.183667 -11.400785  -8.604472  -6.716822  -4.719166 -2.930147
        2002-03-01  -3.841712  -1.114428   0.856563   1.837213   3.480946  4.461936
        2002-04-01 -27.105056 -19.049959 -15.639919 -13.077801 -10.816960 -8.427072
        2002-05-01 -14.407484  -9.260408  -6.635828  -4.936906  -3.012894 -1.392009
        2002-06-01 -29.869803 -21.181518 -17.600453 -14.850385 -12.516204 -9.958874

                       0.7       0.8        0.9
        Date
        2002-02-01 -1.674577  0.364496   6.308721
        2002-03-01  7.037983  9.598789  16.839746
        2002-04-01 -8.153439 -6.502338  -1.522394
        2002-05-01  0.138325  2.285960   8.500008
        2002-06-01 -9.958874 -8.415887  -3.704654
```

Unfortunately, there is no existing function in Python to implement a formal statistical difference test between quantiles as is available other software. However, we can still superficially visualise each regression line in a plot. To do so, we first create a figure object named 1, followed by a **for** loop to plot each quantile. Specifically, we make use of the output **y_hats** of the function **model_paras** where the fitted values for each quantile have been computed. Then, both actual observations **data['ersandp']** and predicted values **y_hats[i]** are assigned to new variables x, y, respectively, each time. To highlight the OLS regression specification (i.e., when quantile is 0.5), we plot it in red while the remaining quantiles are displayed in grey. Finally, the figure will appear (Figure 18) in the window below after setting x, y axis and so on.

```
In [7]: plt.figure(1)
        for i in quantreg_paras.q:
            x = data['ersandp']
            y = y_hats[i]
            if i == 0.50:
                plt.plot(x,y,color='red')
```

```
    else:
        plt.plot(x,y,color='grey')


plt.ylabel('ersandp')
plt.xlabel('erford')
plt.show()
```

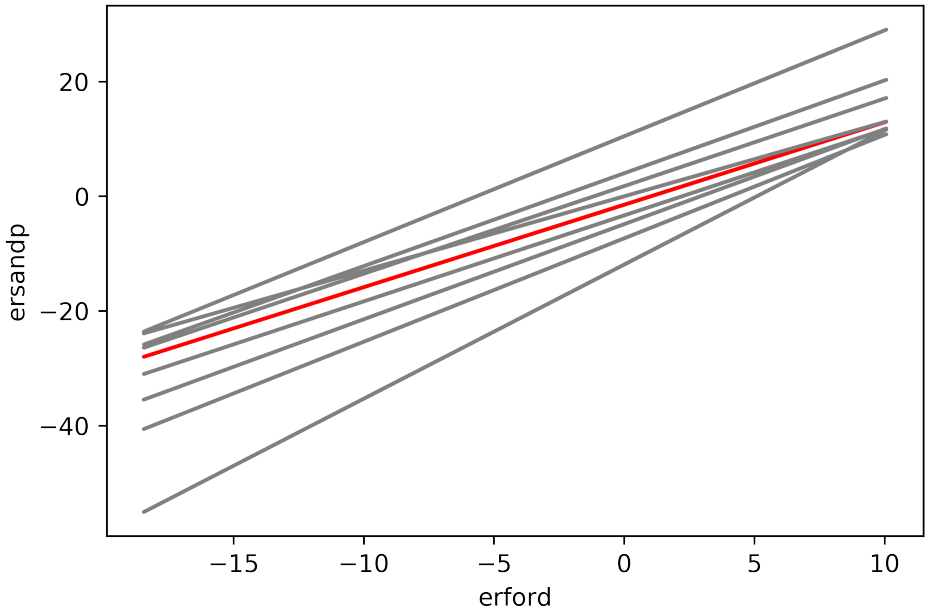


Figure 18: Linear Regression for Different Quantiles

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 in a NoteBook. First we import the 'FRED' Excel workfile which contains US Treasury bill and bond series of various maturities. As usual, we use the Pandas **read_excel** function to achieve this task and print the first five rows to check whether the dataset is imported appropriately. As shown in Out [1], the data contain six US Treasury bill series a follows:
GS3M GS6M GS1 GS3 GS5 GS10
Note that there are multiple ways to perform principal component analysis in Python. Here, we calculate them by designing a function instead of importing a third-party Python module.

```
In [1]: from NumPy import mean,cov,cumsum,dot,linalg,std,sort
        import Pandas as pd
        import matplotlib.pyplot as plt

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                          'QMF Book/book Ran/data files new/Book4e_data/'

        data = pd.read_excel(abspath + 'FRED.xls', index_col=0)

        data.head()
```

Out[1]:

date	GS3M	GS6M	GS1	GS3	GS5	GS10
1990-01-01	7.90	7.96	7.92	8.13	8.12	8.21
1990-02-01	8.00	8.12	8.11	8.39	8.42	8.47
1990-03-01	8.17	8.28	8.35	8.63	8.60	8.59
1990-04-01	8.04	8.27	8.40	8.78	8.77	8.79
1990-05-01	8.01	8.19	8.32	8.69	8.74	8.76

The principal components are calculated as follows: First, we specify the input of the function as **data**, which contains six US Treasury bill series. It is good practice to clarify that the input m-by-n matrix should comprise m rows of observations and n columns of variables. As we have checked the structure of the **data**, it exactly matches the requirement. Next, we need to normalise the sample by subtracting the average of each column and then dividing by the respective standard deviations individually. Again, we use the combination of **apply** and **lambda**, which can easily map the matrix. After that, we calculate the covariance matrix of **M**. Note that the function **cov** from NumPy requires a column vector input, but **M** is a row vector. To fit the input, we transpose the DataFrame **M**. The eigenvalues and eigenvectors then can be computed by the function **linalg.eig**, which also comes from the NumPy module. Additionally, we add the command **latent = sort(latent)[::-1]** to ensure that the eigenvalues are listed in an ascending order. Finally, **latent** and **coeff** are converted to the DataFrame and Series, respectively, and are returned.

```
In [2]: def princomp(data):
            '''
            Computing eigenvalues and eigenvectors of covariance matrix

            Parameters:
            -----------
```

```
        data: the m-by-n DataFrame which corresponds m rows of observations
              and n columns of variables.

        Returns:
        -----------
        coeff: a n-by-n matrix (eigenvectors) where it contains all coefficients
               of principal component for each variable.

        latent: a vector of the eigenvalues
        '''

        M = data.apply(lambda x: (x-mean(x))/std(x) ) # normalize

        # Note: cov inputs require a column vector;
        # That is each row of m represents a variable,
        # and each column is a single observation of all those variables
        # To tackle this, simply add argument rowvar=True
        # or transpose M matrix
        [latent,coeff] = linalg.eig(cov(M.transpose()))
        # attention: latent (eigenvalues) is not always sorted
        latent = sort(latent)[::-1]

        # convert arrays to DataFrame or Series
        coeff = pd.DataFrame(coeff.T, columns=data.columns)
        latent = pd.Series(latent, name='Eigenvalue')

        return coeff, latent
```

Call the function we defined and examine the eigenvectors and eigenvalues. It is evident that there is a great deal of common variation in the series, since the first principal component captures over 96% of the variation in the series and the first two components capture 99.8% [see In [5], [6]] and Figure 19. 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 longest 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).

```
In [3]: coeff, latent = princomp(data)

In [4]: coeff

Out[4]:        GS3M      GS6M       GS1       GS3       GS5      GS10
        0  0.407769  0.409146  0.411719  0.414379  0.409883  0.396356
        1  0.417793  0.391519  0.293202 -0.091793 -0.361692 -0.668541
        2 -0.468801 -0.152028  0.231295  0.588798  0.293663 -0.520283
        3 -0.537538  0.192442  0.624653 -0.154963 -0.414705  0.296368
        4 -0.307086  0.506547 -0.082492 -0.524346  0.580709 -0.173614
        5  0.236960 -0.602142  0.542243 -0.417407  0.325168 -0.085348
```

```
In [5]: perc_lat = cumsum(latent)/sum(latent)
        print(perc_lat)

0    0.965461
1    0.997919
2    0.999615
3    0.999924
4    0.999976
5    1.000000
Name: Eigenvalue, dtype: float64
```

```
In [6]: plt.figure(1)
        plt.stem(range(len(perc_lat)),perc_lat,'--b')
        plt.ylabel('Percentage of Eeigenvalues')
        plt.xlabel('The number of components')
        plt.show()
```

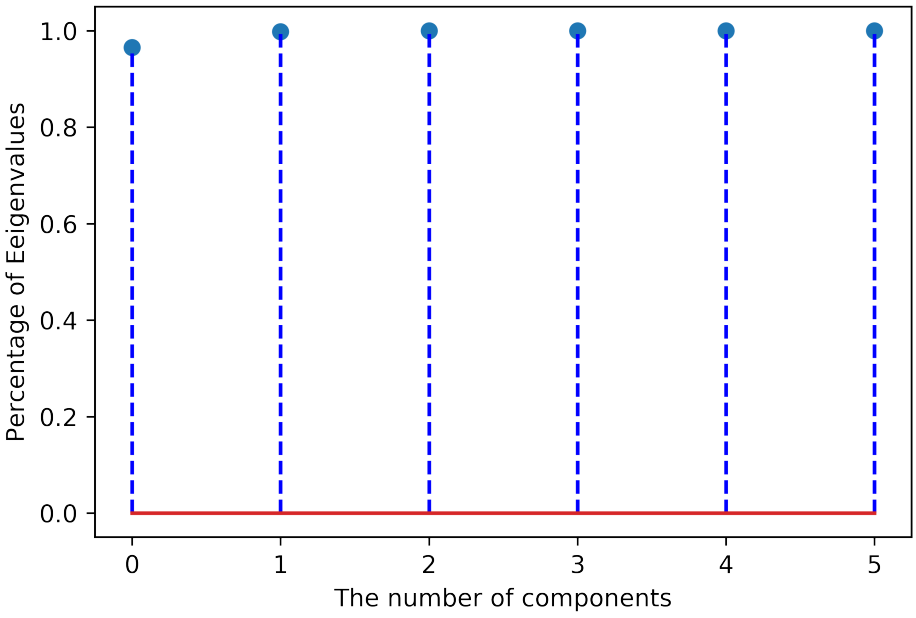


Figure 19: Percentage of Eigenvalues Attributable to Each Component

10 Diagnostic testing

10.1 Testing for heteroscedasticity

Reading: Brooks (2019, section 5.4)
In this example we will undertake a test for heteroscedasticity in Python, using the 'macro.pickle' workfile. We will inspect the residuals of an 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 to reproduce the regression results. The simplest way is to re-estimate the regression by typing the following command in In[1] and In[2]

```
In [1]: import pickle
        import statsmodels.formula.api as smf
        import statsmodels.stats.api as sms
        import matplotlib.pyplot as plt
        from statsmodels.compat import lzip

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'macro.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data = data.dropna() # drop the missing values for some columns

        formula = 'ermsoft ~ ersandp + dprod + dcredit + \
                            dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit()
```

This time, we are less interested in the coefficient estimates reported in the regression summary table, but we focus on the properties of the residuals from this regression. To get a first impression of the characteristics of the residuals, we want to plot them. To obtain the residuals, we use the command **results.resid**, which allows us to access the residuals of regression specification from the instance **results**. To plot this series, we then create a figure and plot the series by the **plot**. The X and Y axes are further options to be set up depending on what the user requires. A final printed image can be seen In [3].

Let us examine the patterns in the residuals over time (see Figure 20). If the residuals of the regression have a 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 a formal statistical test.

```
In [2]: plt.figure(1)
        plt.plot(results.resid)
        plt.xlabel('Date')
        plt.ylabel('Residuals')
        plt.grid(True)
        plt.show()
```

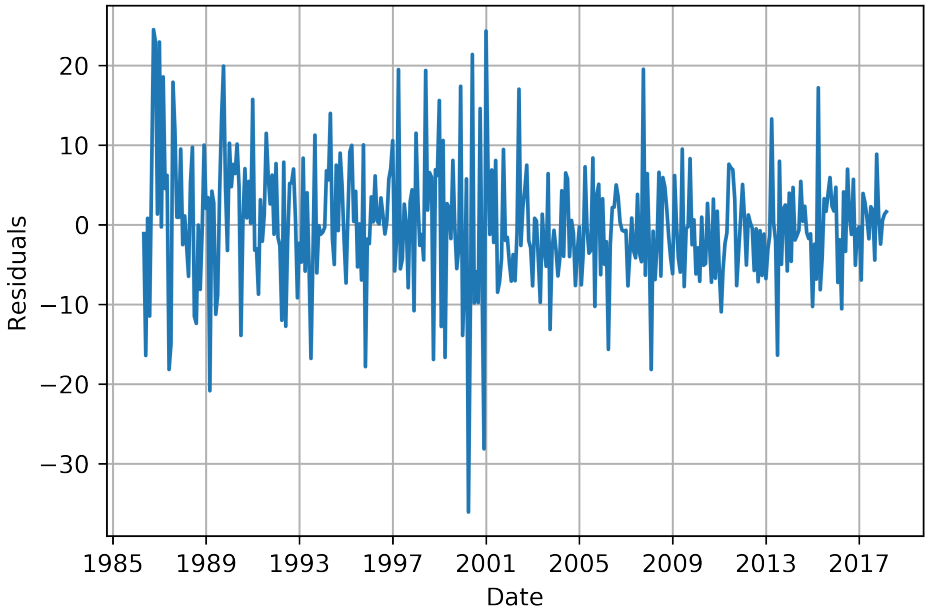


Figure 20: Time-series Plot of Residuals

To do so, we select the function **het_breuschpagan** from the Statsmodel library. The aim of this function is to test the hypothesis that the residual variance does not depend on functions (higher order powers or multiplicative combinations) of the explanatory variables, thus requiring two essential inputs: 'residuals' and 'exogenous variables'.

There are four outputs in total produced by this function: a Lagrange multiplier statistic, p-value, F-test statistic value and F-test p-value. Note that the **het_breuschpagan** tests the hypothesis in two ways. The first method is to employ the generic formula for an LM test using n*R^2 (the number of observations times the auxiliary regression), while the F-statistic is preferable when samples are small or moderately large since otherwise the test exaggerates the statistical significance. To better present the two test results, we finally create a list with four names and link them to each statistic.

As you can see from the test statistics and *p*-values, both tests lead to the conclusion that there does not seem to be a serious problem of heteroscedastic errors for our APT-style model with the *p*-values both being 0.87, respectively.

```
In [3]: # breusch-pagan heteroskedasticity test
        name = ['Lagrange multiplier statistic', 'p-value',
                'f-value', 'f p-value']
        test = sms.het_breuschpagan(results.resid, results.model.exog)
        lzip(name, test)
```

```
Out[3]: [('Lagrange multiplier statistic', 3.1606601504201448),
         ('p-value', 0.86975267437865456),
         ('f-value', 0.4457702552712059),
         ('f p-value', 0.87290034041883569)]
```

10.2 Using White's modified standard error estimates

Reading: Brooks (2019, subsection 5.4.3)

We can specify to estimate the regression with heteroscedasticity-robust standard errors in Python. When we create the regression specification by the command **smf.ols(formula, data)**, we usually follow the function **fit** to produce the result instance. So far, we have only focused on the regression model itself that specifies the dependent and independent variables. If we move to usage of the **fit** function, we are presented with different options for adjusting the standard errors.

```
In [1]: import statsmodels.formula.api as smf
        import pickle

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                          'QMF Book/book Ran/data files new/Book4e_data/'

        with open(abspath + 'macro.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data = data.dropna() # drop the missing values for some columns
```

In order to obtain standard errors that are robust to heteroscedasticity, we select the argument **cov_type='HC1'**. Obviously, there are a number of options you can select. For more information on the different standard error adjustments, refer to the Statsmodel documentation.[22] Comparing the regression output for our APT-style model using robust standard errors with that using ordinary standard errors, we find that the changes in significance are only marginal, as shown in the output below.

```
In [2]: formula = 'ermsoft ~ ersandp + dprod + dcredit + \
                           dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit(cov_type='HC1')
        print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                ermsoft   R-squared:                       0.345
Model:                            OLS   Adj. R-squared:                  0.333
Method:                 Least Squares   F-statistic:                     29.89
Date:                Thu, 09 Aug 2018   Prob (F-statistic):           9.27e-33
Time:                        11:27:38   Log-Likelihood:                -1328.3
No. Observations:                 383   AIC:                             2673.
Df Residuals:                     375   BIC:                             2704.
Df Model:                           7
Covariance Type:                  HC1
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      1.3260      0.459      2.888      0.004       0.426       2.226
ersandp        1.2808      0.093     13.778      0.000       1.099       1.463
```

[22] The detail of how to implement each standard error adjustment in Python is listed on https://www.Statsmodels.org/dev/generated/Statsmodels.regression.linear_model.RegressionResults.html.

```
dprod          -0.3030      0.635     -0.478      0.633      -1.547       0.941
dcredit        -0.0254      0.021     -1.219      0.223      -0.066       0.015
dinflation      2.1947      1.307      1.679      0.093      -0.367       4.756
dmoney         -0.0069      0.011     -0.630      0.528      -0.028       0.014
dspread         2.2601      3.428      0.659      0.510      -4.458       8.978
rterm           4.7331      1.727      2.741      0.006       1.349       8.117
==============================================================================
Omnibus:                       21.147   Durbin-Watson:                   2.097
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               63.505
Skew:                          -0.006   Prob(JB):                     1.62e-14
Kurtosis:                       4.995   Cond. No.                         293.
==============================================================================

Warnings:
[1] Standard Errors are heteroscedasticity robust (HC1)
```

Of course, only the standard errors have changed and the parameter estimates remain identical to those estimated before. The heteroscedasticity-consistent standard errors are smaller for most variables, resulting in t-ratios growing in absolute value and p-values being smaller. The main changes in the conclusions reached are that the difference in the consumer credit variable ('dcredit'), which was previously significant only at the 10% level, is now significant at 5%, and the unexpected inflation and change in industrial production variables are now significant at the 10% level.

10.3 The Newey-West procedure for estimating standard errors

Reading: Brooks (2019, sub-section 5.5.7)

In this subsection, we will apply the Newey-West procedure for estimating heteroscedasticity and autocorrelation robust standard errors in Python. Similar to implementing robust standard error adjustments by an argument in the function **fit**, the Newey-West procedure only requires us to change the argument. To access this command, we simply type the argument **cov_type='HAC', cov_kwds='maxlags':6,'use_correction':True** . Of course, before implementing the robust standard error adjustment, we are first required to define the dependent variable and the independent variables in the formula and to create the regression specification.

As can be seen, after the command we typed, we are asked to specify the maximum number of lag to consider in the autocorrelation structure, i.e., we need to manually input the maximum number of lagged residuals to be considered for inclusion in the model. There might be different economic motivations for choosing the maximum lag length, depending on the specific analysis one is undertaking. In our example, we decide to include a maximum lag length of six, implying that we assume that the potential autocorrelation in our data does not go beyond a window of six months.[23] By hitting **SHIFT** and **ENTER**, the following regression results appear in the output.

```
In [1]: import statsmodels.formula.api as smf
        import pickle

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                          'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'macro.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data = data.dropna() # drop the missing values for some columns

In [2]: formula = 'ermsoft ~ ersandp + dprod + dcredit + \
                           dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit(cov_type='HAC',
                                             cov_kwds={'maxlags':6,'use_correction':True})
        print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                ermsoft   R-squared:                       0.345
Model:                            OLS   Adj. R-squared:                  0.333
Method:                 Least Squares   F-statistic:                     24.93
Date:                Thu, 09 Aug 2018   Prob (F-statistic):           6.73e-28
Time:                        11:28:44   Log-Likelihood:                -1328.3
No. Observations:                 383   AIC:                             2673.
Df Residuals:                     375   BIC:                             2704.
Df Model:                           7
Covariance Type:                  HAC
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
```

[23] Note that if we were to specify zero in the 'maxlags' argument, the Newey-West adjusted standard errors would be the same as the robust standard errors introduced in the previous section.

```
------------------------------------------------------------------------------
Intercept      1.3260      0.503      2.638      0.008       0.341       2.311
ersandp        1.2808      0.100     12.822      0.000       1.085       1.477
dprod         -0.3030      0.522     -0.581      0.561      -1.325       0.719
dcredit       -0.0254      0.022     -1.135      0.256      -0.069       0.018
dinflation     2.1947      1.314      1.671      0.095      -0.380       4.769
dmoney        -0.0069      0.011     -0.628      0.530      -0.028       0.015
dspread        2.2601      2.842      0.795      0.426      -3.310       7.830
rterm          4.7331      1.759      2.692      0.007       1.286       8.180
==============================================================================
Omnibus:                       21.147   Durbin-Watson:                   2.097
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               63.505
Skew:                          -0.006   Prob(JB):                     1.62e-14
Kurtosis:                       4.995   Cond. No.                         293.
==============================================================================

Warnings:
[1] Standard Errors are heteroscedasticity and autocorrelation robust (HAC) using 6 lags
and with small sample correction
```

10.4 Autocorrelation and dynamic models

Reading: Brooks (2019, subsections 5.5.7 – 5.5.11)
In this section, we want to apply different tests for autocorrelation in Python using the APT-style model of the previous section (the 'macro.pickle' workfile).

The simplest test for autocorrelation is due to Durbin and Watson (1951). It is a test for first-order autocorrelation - i.e., it tests only for a relationship between an error and its immediately previous value. To access the Durbin-Watson test, we access the function **durbin_watson** via the module **statsmodels.stats.api** and generate a new variable **residuals** by the command **results.resid**.

```
In [1]: import statsmodels.formula.api as smf
        import statsmodels.stats.api as sms
        from statsmodels.compat import lzip
        import pickle

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                             'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'macro.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data = data.dropna() # drop the missing values for some columns
```

```
In [2]: # durbin_watson
        formula = 'ermsoft ~ ersandp + dprod + dcredit + \
                             dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit()

        residuals = results.resid
        sms.durbin_watson(residuals)
```

```
Out[2]: 2.0973940504299913
```

The value of the *DW* statistic is 2.097. 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 more general than *DW* and allows us to test for higher order autocorrelation. In Python, the Breusch-Godfrey test can be conducted by the same module **statsmodels.stats.api**. There are two inputs for this test: we need to feed the function with the **results** instance created by the OLS specification beforehand and employ **10** lags in the test. However, unlike the Durbin-Watson test, which only produces one statistic, there are four outputs for the Breusch-Godfrey test. This is because the hypothesis is tested in two different ways. Consistent with the example above, we create a list with the name of the output in order to better present the results. The final results shall appear as below.

```
In [3]: name = ['Lagrange multiplier statistic', 'p-value',
                'f-value', 'f p-value']
        results1 = sms.acorr_breusch_godfrey(results, 10)
        lzip(name, results1)
```

```
Out[3]: [('Lagrange multiplier statistic', 4.766591436782349),
         ('p-value', 0.9062145800242255),
         ('f-value', 0.45998207324796836),
         ('f p-value', 0.91501799815728901)]
```

10.5 Testing for non-normality

Reading: Brooks (2019, section 5.7)

One of the most commonly applied tests for normality is the Bera-Jarque (BJ) test. 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, i.e. the 'residuals' variable that we created in subsection 10.4.

```
In [1]: import statsmodels.formula.api as smf
        import statsmodels.stats.api as sms
        from statsmodels.compat import lzip
        import matplotlib.pyplot as plt
        import pickle

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'macro.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data = data.dropna() # drop the missing values for some columns
```

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 of whether the residuals might be normally distributed. 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 first generate an OLS regression outcome instance with the variable named **results** and access the residual series by the command **results.resid**. We next are going to plot the 'residuals'. Specifically, we first create a figure object by the module **matplotlib.pyplot**. Plotting the series by the function **hist** and setting up additional parameters such as the number of bins, edgecolour, linewidth, x and y axis and so on. The histogram can be seen in In [2].

```
In [2]: formula = 'ermsoft ~ ersandp + dprod + dcredit + \
                            dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit()
        residuals = results.resid

        plt.figure(1)
        plt.hist(residuals,20,edgecolor='black',linewidth=1.2)
        plt.xlabel('Residuals')
        plt.ylabel('Density')
        plt.show()
```

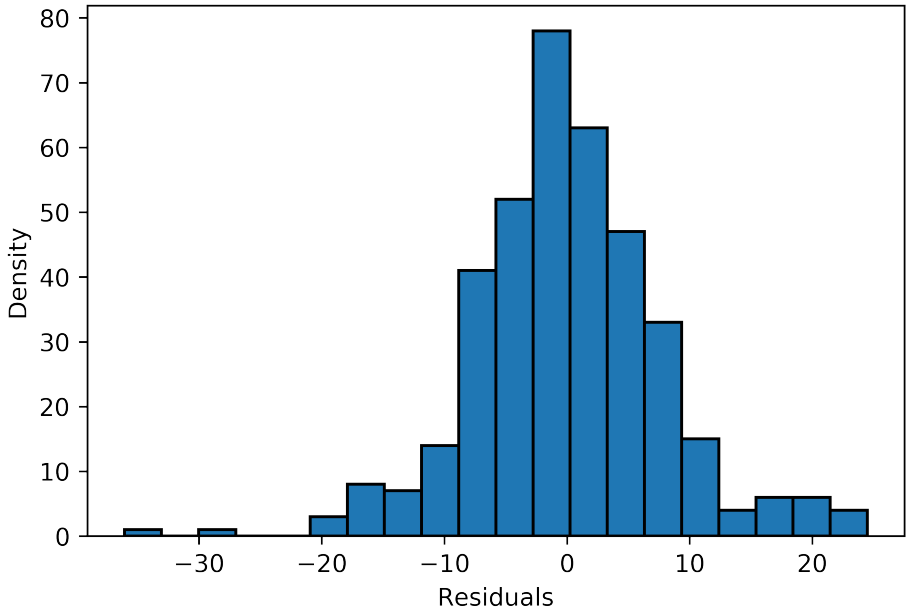


Figure 21: Histogram of Residuals

Looking at the histogram plot (Figure 21), we see that the distribution of the residuals roughly resembles a bell-shape although we also find that there are some large negative outliers which might lead to a considerable negative skewness. We could increase the number of bins or lower the width of the bins in order to obtain 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 normality test. The standard test for the normality of a data series in Python is the BJ test. To do so, we feed the variable 'residuals' into the function **jarque_bera** and zip the output's names with outcomes. The result appear in the output window below (see Out [3]).

```
In [3]: name = ['Jarque-Bera', 'Chi^2 two-tail prob.', 'Skew', 'Kurtosis']
        test = sms.jarque_bera(residuals)
        lzip(name, test)
```

```
Out[3]: [('Jarque-Bera', 63.50547267158968),
         ('Chi^2 two-tail prob.', 1.6216675409916346e-14),
         ('Skew', -0.00561267747998391),
         ('Kurtosis', 4.99482560590665)]
```

Python reports the BJ statistic and χ^2 two-tailed p-value for the test that the residuals are overall normally distributed, i.e., that both the kurtosis and the skewness are those of a normal distribution.

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 –25%. What does the non-normality of residuals imply for the inferences we make about coefficient estimates? Generally speaking, it could mean that these inferences could be wrong, although the sample is probably large enough that we need be less concerned than we would with a smaller sample.

10.6 Dummy variable construction and use

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 already generated a data series containing the residuals of the Microsoft regression. Let us now create a series of the fitted values. For this, we use the function **fittedvalues** from the **results** instance. We name the variable **y_fitted** and define that it will contain the linear prediction for the Microsoft regression. The variable **residuals** is generated by the command {results.resid}.

```
In [1]: import statsmodels.formula.api as smf
        import matplotlib.pyplot as plt
        import NumPy as np
        import pickle

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'macro.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data = data.dropna() # drop the missing values for some columns

In [2]: # regression
        formula = 'ermsoft ~ ersandp + dprod + dcredit + \
                             dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit()
        y_fitted = results.fittedvalues
        residuals = results.resid
```

In order to plot both the residuals and fitted values in one times-series graph, we import the **matplotlib.pyplot** module. As usual, we create a figure object and plot two separate series in two commands. To be able to clearly illustrate each series in the chart, we add the **label** argument to represent each data series. Then we set up the x and y axis, grid, legend and so on. Finally, type the command **plt.show()** and the two series appear in figure 22.

```
In [3]: plt.figure(1)
        plt.plot(residuals, label='resid')
        plt.plot(y_fitted, label='linear prediction')
        plt.xlabel('Date')
        plt.ylabel('Residuals')
        plt.grid(True)
        plt.legend()
        plt.show()
```

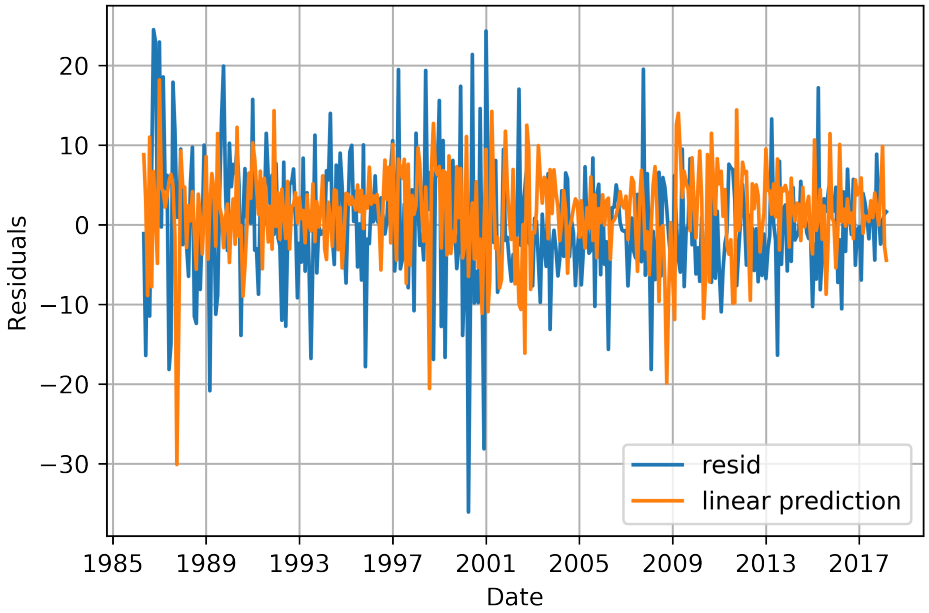


Figure 22: Regression Residuals and Fitted Series

From the graph, it can be seen that there are several large (negative) outliers, but the largest all occur in 2000. These 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 considerably, 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 when the biggest outliers were realised, it is probably easiest to just examine a table of values for the residuals. This can be done by applying the Pandas built-in function **nsmallest** and specifying the number of smallest values to pick up. Hitting **SHIFT** and **ENTER** leads Python to output the two smallest values from the given series, as shown in Out [4] below.

```
In [4]: residuals.nsmallest(2)

Out[4]: Date
        2000-04-01   -36.075347
        2000-12-01   -28.143156
        dtype: float64
```

It is evident that the two most extreme residuals were in April (–36.075) and December 2000 (–28.143). 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 and December 2000. 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 April 2000 dummy first, we generate a series called 'APR00DUM' into **data** DataFrame. To precisely assign the value 1 on the date of April 2000 while keeping the rest of the series as zeros, we

employ the built-in function **where** from the NumPy module; there are three parameters are required as inputs: condition, x, and y, respectively. The first argument would be the condition, since the aim of this function is to return elements, either from the series x or y, depending on the condition. In other words, the function will yield the value of x when the condition is true, otherwise it will yield y. In this setting, we set up the argument **data.index == '2000-4-1'** since the returned elements would take the value 1 only when the index value of the **data** DataFrame is April 2000.

We repeat the process above to create another dummy variable called 'DEC00DUM' that takes the value 1 in December 2000 and zero elsewhere.

Let us now rerun the regression to see whether the results change once we have removed 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 written in the formula statement. Repeating the regression specification commands, the output of this regression should look as follows.

```
In [5]: data['APR00DUM'] = np.where(data.index == '2000-4-1', 1, 0)
        data['DEC00DUM'] = np.where(data.index == '2000-12-1', 1, 0)

        # regression
        formula = 'ermsoft ~ ersandp + dprod + dcredit + \
                             dinflation + dmoney + dspread + rterm + \
                             APR00DUM + DEC00DUM'
        results = smf.ols(formula, data).fit()
        print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 ermsoft   R-squared:                       0.406
Model:                             OLS   Adj. R-squared:                  0.392
Method:                  Least Squares   F-statistic:                     28.33
Date:                 Thu, 09 Aug 2018   Prob (F-statistic):           2.22e-37
Time:                         11:31:57   Log-Likelihood:                -1309.7
No. Observations:                  383   AIC:                             2639.
Df Residuals:                      373   BIC:                             2679.
Df Model:                            9
Covariance Type:             nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      1.4198      0.454      3.125      0.002       0.526       2.313
ersandp        1.2539      0.090     13.897      0.000       1.076       1.431
dprod         -0.3211      0.705     -0.456      0.649      -1.707       1.065
dcredit       -0.0157      0.026     -0.603      0.547      -0.067       0.035
dinflation     1.4421      1.215      1.187      0.236      -0.946       3.830
dmoney        -0.0057      0.015     -0.383      0.702      -0.035       0.024
dspread        1.8693      3.955      0.473      0.637      -5.908       9.647
rterm          4.2642      1.641      2.599      0.010       1.038       7.490
APR00DUM     -37.0288      7.576     -4.888      0.000     -51.926     -22.132
DEC00DUM     -28.7300      7.546     -3.807      0.000     -43.569     -13.891
==============================================================================
Omnibus:                       20.084   Durbin-Watson:                   2.088
```

```
Prob(Omnibus):                  0.000   Jarque-Bera (JB):                28.877
Skew:                           0.404   Prob(JB):                     5.36e-07
Kurtosis:                       4.076   Cond. No.                          560.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

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.[24] 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 and the significances improve considerably. 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.

[24] 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.

10.7 Multicollinearity

Reading: Brooks (2019, section 5.8)

Let us assume that we would like to test for multicollinearity in the Microsoft regression ('macro.pickle' workfile). To generate a correlation matrix in Python, we apply the Pandas built-in function **corr** followed by the DataFrame **data**. In the command for indexing Pandas columns, we enter the list of regressors (not including the regressand or the S&P500 returns), as in In [2].

```
In [1]: import pickle

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'macro.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data = data.dropna() # drop the missing values for some columns

In [2]: data = data[['dprod','dcredit','dinflation','dmoney','dspread','rterm']]
        data.corr()
```

Out[2]:

	dprod	dcredit	dinflation	dmoney	dspread	rterm
dprod	1.000000	0.094273	-0.143551	-0.052514	-0.052756	-0.043751
dcredit	0.094273	1.000000	-0.024604	0.150165	0.062818	-0.004029
dinflation	-0.143551	-0.024604	1.000000	-0.093571	-0.227100	0.041606
dmoney	-0.052514	0.150165	-0.093571	1.000000	0.170699	0.003801
dspread	-0.052756	0.062818	-0.227100	0.170699	1.000000	-0.017622
rterm	-0.043751	-0.004029	0.041606	0.003801	-0.017622	1.000000

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. Both are probably sufficiently small in absolute value 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 our Microsoft regression, we need to import the **reset_ramsay** function from the **statsmodels.stats.outliers_influence** module. This function requires the regression results instance, and thus we need to create an OLS regression specification first by typing the following commands as usual.

```
In [1]: from statsmodels.stats.outliers_influence import reset_ramsey
        import statsmodels.formula.api as smf
        import pickle

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'macro.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data = data.dropna() # drop the missing values for some columns

In [2]: formula = 'ermsoft ~ ersandp + dprod + dcredit + \
                            dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit()

        reset_ramsey(results,degree=4)

Out[2]: <class 'Statsmodels.stats.contrast.ContrastResults'>
        <F test: F=array([[ 0.9937673]]), p=0.39574412394082514, df_denom=372, df_num=3>
```

The result of the test appears in Out [2]. Take some time to examine the result statistics. Based on the F-statistic having four degrees of freedom, we can assume that Python included three higher order terms of the fitted values in the auxiliary regression. With an F-value of 0.9937 and a corresponding p-value of 0.3957, the RESET test results imply 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 that we can apply: known structural break tests and unknown structural break tests. The former requires a pre-specific datetime to split the sample data, while the latter examines the parameter stability without any priors as to which variables might be subject to a structural break and which might not. In Python, the Statsmodels library has incomplete functions in terms of regression diagnostics, particularly for parameter stability tests. The lack of these is unfortunate, but leaves us an opportunity to explore these tests in detail by designing our own functions in Python. Let us first have a go at the Chow test.

As usual, we firstly import several necessary libraries and re-load the 'macro.pickle' work file beforehand (In [1]). We next define a custom function **get_rss** where the residual sum of squares (RSS), degrees of freedom and the number of observations are generated from the given regression formula. Note that it is not possible to conduct a Chow test or a parameter stability test when there are outlier dummy variables in the regression. Thus, we have to ensure that the estimation that we run is the Microsoft regression omitting the APR00DUM and DEC00DUM dummies from the list of independent variables.[25] This occurs because when the sample is split into two parts, the dummy variable for one of the parts will have values of zero for all of the observations, which would thus cause perfect multicollinearity with the column of ones that is used for the constant term. To obtain the results, a regression instance **results** is constructed as usual with three additional commands: **rss = (results.resid*2).sum()**, **N = results.nobs** and **K = results.df_model**. Finally, we complete the function by returning the results.

```
In [1]: import statsmodels.stats.api as sms
        import statsmodels.formula.api as smf
        from statsmodels.compat import lzip
        import pickle
        import Pandas as pd
        import matplotlib.pyplot as plt

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'macro.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data = data.dropna() # drop the missing values for some columns

In [2]: def get_rss(data):
            '''
            inputs:
                data: a Pandas DataFrame of independent and dependent variable
            outputs:
                rss: the sum of residuals
                N: the observations of inputs
                K: total number of parameters
            '''
            formula = 'ermsoft ~ ersandp + dprod + dcredit + \
```

[25] The corresponding formula for this regression is: 'ermsoft ∼ ersandp + dprod + dcredit + dinflation + dmoney + dspread + rterm'.

```
                                dinflation + dmoney + dspread + rterm'
    results = smf.ols(formula, data).fit()
    rss = (results.resid**2).sum() # obtain the residuals sum of square
    N = results.nobs
    K = results.df_model
    return rss, N, K
```

Let us assume that we want to test whether a breakpoint occurred in January 1996. Thus, we specify the hypothesised break date '1996-01-01' and split our sample into **data1** and **data2**. To calculate the Chow statistic given its formula as below, the following pre-defined function **get_rss** is implemented for three different samples individually. With all parameters in hand, readers can easily compute the final result given the code presented in cell [3] below.

$$Result = \frac{(RSS_{total} - (RSS_1 + RSS_2))/K_{total}}{(RSS_1 + RSS_2)/(N_1 + N_2 - 2 * K_{total})} \tag{2}$$

```
In [3]: # split samples
        data1 = data[:'1996-01-01']
        data2 = data['1996-01-01':]

        # get rss of whole sample
        RSS_total, N_total, K_total = get_rss(data)
        # get rss of the first part of sample
        RSS_1, N_1, K_1 = get_rss(data1)
        # get rss of the second part of sample
        RSS_2, N_2, K_2 = get_rss(data2)

        nominator = (RSS_total - (RSS_1 + RSS_2)) / K_total
        denominator = (RSS_1 + RSS_2) / (N_1 + N_2 - 2*K_total)

        result = nominator/denominator
```

Note that the test statistic follows the F distribution with $\{K_{total}\}$ k and $\{N_1 + N_2 - 2K_{total}\}$ degrees of freedom, which is 7 and 370 in this case. If readers compare the statistic with the corresponding value from the F distribution tables at the 10 % significance level (i.e., 1.71672), it is easy to reach the conclusion that we can reject the null hypothesis that the parameters are constant across the two subsamples, i.e., before and after '1996-01-01'.

```
In [4]: result
```

```
Out[4]: 1.9894610789615745
```

It is often the case that the date when the structural break occurs is not known in advance. Python thus offers a test that does not require us to specify the break date but tests for each possible break date in the sample. This test can be accessed via the Statsmodels library **statsmodels.stats.api**, which is known as **breaks_cusumolsresid**. The implementation of this function is easy, as only two inputs are required, i.e., the OLS regression residuals and the degrees of freedom. The results will appear in the following output window if readers execute the code in In [5].

```
In [5]: formula = 'ermsoft ~ ersandp + dprod + dcredit + \
                           dinflation + dmoney + dspread + rterm'
        results = smf.ols(formula, data).fit()

        name = ['test statistic', 'pval', 'crit']
        test = sms.breaks_cusumolsresid(olsresidual = results.resid,\
                                        ddof = results.df_model)
        lzip(name, test)

Out[5]: [('test statistic', 1.5352572458140492),
         ('pval', 0.017937116553892265),
         ('crit', [(1, 1.63), (5, 1.36), (10, 1.22)])]
```

The results give three rows of statistics. The first two rows present the test statistics and corresponding p-value, while the last row provides 1%, 5% and 10% significance levels with their critical values, respectively. 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 does have a structural break for any possible break date in the sample.

Another way of testing whether the parameters are stable with respect to any break dates is to a test based on recursive estimation. Unfortunately, there is no built-in function in Python that automatically produces plots of the recursive coefficient estimates together with standard error bands. In order to visually investigate the extent of parameter stability, we have to create a function where recursive estimations can be performed under iteration. Then we can plot these data series.

To do so, we first create a function called **recursive_reg**. In the inputs, we set three parameters that need to be pre-specified: variable name, the iteration number and initial sample size. To obtain the recursively estimated coefficients and standard errors, we need to construct the regression instance by slicing the sample data. This can be easily achieved by the command **data.iloc[:i+interval]**. Specifically, the regression sample would be the first 11 data points if the input i takes the value 1. If i then becomes 2, the sample size increases to 12 observations, and so on. After that, two outputs can be accessed by the command **coeff = results.params[variable]** and **se = results.bse[variable]**. The function finally ends up returning the output.

```
In [6]: def recursive_reg(variable, i, interval):
            '''
            Parameters:
            -----------
                variable: the string literals of a variable name in regression
                          formula.
                i: the serial number of regression.
                interval: the number of consective data points in initial sample

            Returns:
            -----------
                coeff: the coefficient estimation of the variable
                se: the standard errors of the variable
            '''
            formula = 'ermsoft ~ ersandp + dprod + dcredit + \
                               dinflation + dmoney + dspread + rterm'
```

```
        results = smf.ols(formula, data.iloc[:i+interval]).fit()
        coeff = results.params[variable]
        se = results.bse[variable]

        return coeff, se
```

We implement this function under a loop where there are 373 iterations to perform given 11 initial sample observations. Obviously, you can always change the initial number of data points if desired and revise the number of iterations accordingly. To save the results, we create an empty list **parameters** as usual and update it by the function **append**. In order to better present it, the list is then converted to a DataFrame with column names and a time-series index. Note that our **parameters** list only contains 373 estimations, and therefore the corresponding index column should drop the last 11 points.

In order to visually investigate the extent of parameter stability over the recursive estimations, it is best to generate a time-series plot of the recursive estimates. Assume that we would like to generate such a plot for the recursive estimates of the **ersandp**. We would like to plot the actual recursive coefficients together with their standard error bands. So first, we need to obtain data series for the standard error bands. We generate two series: one for a deviation of two standard errors above the coefficient estimate ($\beta_{ersandp} + 2 * SE$) and one for a deviation of two standard errors below the coefficient estimate ($\beta_{ersandp} - 2 * SE$). To do so, we use the following two commands to generate the two series:
parameters['ersandp + 2se'] = parameters['coeff'] + 2parameters['se']
parameters['ersandp - 2se'] = parameters['coeff'] - 2parameters['se']

```
In [7]: parameters = []
        for i in range(373):
            coeff, se = recursive_reg('ersandp', i, 11)

            parameters.append((coeff,se))

        parameters = pd.DataFrame(parameters, columns=['coeff','se'],\
                                  index = data[:-10].index)

        parameters['ersandp + 2*se'] = parameters['coeff'] + 2*parameters['se']
        parameters['ersandp - 2*se'] = parameters['coeff'] - 2*parameters['se']
```

Once we have generated the new variables, we can plot them together with the actual recursive coefficients of 'ersandp'. We create a figure object named **1** and plot three different series, which are the recursive coefficient estimates for 'ersandp' ($\beta_{ersandp}$), the upper band ($\beta_{ersandp} + 2 * SE$) and the lower band ($\beta_{ersandp} - 2 * SE$). We label each series and set two bands as dashed linespt}. Before generating the final figure, we set up the optional arguments including the x axis, display grid line and legend. Pressing **ENTER** and **SHIFT** leads Python to generate the graph (Figure 23) in the following.

```
In [8]: plt.figure(1)
        plt.plot(parameters['coeff'], label=r'$\beta_{ersandp}$')
        plt.plot(parameters['ersandp + 2*se'], label=r'$\beta_{ersandp} + 2*SE$',\
                 linestyle='dashed')
        plt.plot(parameters['ersandp - 2*se'], label=r'$\beta_{ersandp} - 2*SE$',\
                 linestyle='dashed')
```

```
plt.xlabel('Date')
plt.grid(True)
plt.legend()
plt.show()
```

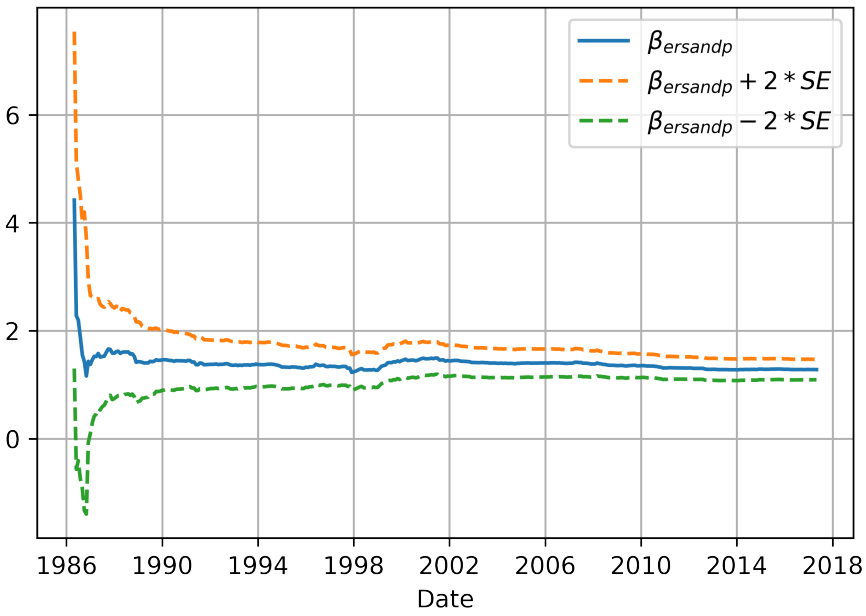


Figure 23: Plot of the Parameter Stability Test

What do we observe from the graph? 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 as the sample size used for estimation grows. 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.

11 Constructing ARMA models

Reading: Brooks (2019, sections 6.4 – 6.7)

Getting started

This example uses the monthly UK house price series which was already incorporated in Python as an example in section 2. In this section, we re-load the data into the NoteBook. Recall the procedure of importing data from an Excel workfile and generating a new variable 'dhp', we type the command as follows (In [1]). To facilitate calculations, we save the data by pickle module for future usage (In [2]).

There are a total of 326 monthly observations running from February 1991 (recall that the January observation was 'lost' in constructing the lagged value) to March 2018 for the percentage change in house price series. 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.

```
In [1]: import Pandas as pd
        import statsmodels.tsa.api as smt
        import pickle

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        data = pd.read_excel(abspath + 'UKHP.xls', index_col=0)
        data['dhp'] = data['Average House Price'].\
                            transform(lambda x : (x - x.shift(1))/x.shift(1)*100)
        data = data.dropna()
        data.head()
```

```
Out[1]:             Average House Price       dhp
        Month
        1991-02-01         53496.798746  0.838950
        1991-03-01         52892.861606 -1.128922
        1991-04-01         53677.435270  1.483326
        1991-05-01         54385.726747  1.319533
        1991-06-01         55107.375085  1.326908
```

```
In [2]: with open(abspath + 'UKHP.pickle', 'wb') as handle:
            pickle.dump(data, handle)
```

Estimating autocorrelation coefficients

To generate a table of autocorrelations, partial correlations and related test statistics, we import the sub-module **statsmodels.tsa.api** from Statsmodels and give it a short name, **smt**. This sub-module provides various time-series analysis functions including acf and pacf, which would be conducted in the following paragraph. To apply the acf function, we type the commands **acf,q,pval = smt.acf(data['dhp'],nlags=12,qstat=True)**. As can be seen, the function requires several input parameters: time-series data, thenumber of lags for autocorrelation and the boolean argument of returning

a Ljung-Box Q statistic for each autocorrelation coefficient.[26] In this case, we input the 'dhp' variable and use **12** lags as the specified number of autocorrelations. The function returns three series, which are all stored in the corresponding variables: acf, q, pval. Likewise, we type a similar command to calculate the pacf. We then create a new DataFrame **correlogram** and store all four results.[27]

```
In [3]: acf,q,pval = smt.acf(data['dhp'],nlags=12,qstat=True)
        pacf = smt.pacf(data['dhp'],nlags=12)

        correlogram = pd.DataFrame({'acf':acf[1:],
                                    'pacf':pacf[1:],
                                    'Q':q,
                                    'p-val':pval})
        correlogram
```

Out[3]:

	Q	acf	p-val	pacf
0	42.056495	0.357530	8.867420e-11	0.358630
1	99.749962	0.418110	2.185601e-22	0.335185
2	117.435756	0.231136	2.751958e-25	0.016514
3	128.484575	0.182406	8.212175e-27	-0.019309
4	133.803012	0.126357	3.709054e-27	0.005922
5	139.502089	0.130596	1.276524e-27	0.055526
6	140.740725	0.060789	3.555650e-27	-0.030376
7	143.867517	0.096431	3.719180e-27	0.036471
8	152.555817	0.160491	2.604935e-28	0.148973
9	158.870433	0.136606	5.542515e-29	0.040527
10	180.862084	0.254530	6.804227e-33	0.153693
11	211.377841	0.299351	1.451072e-38	0.187351

It is clearly evident that the series is quite persistent – surprisingly so given that it is already in percentage change form. The autocorrelation function dies away rather slowly. The numerical values of the autocorrelation and partial autocorrelation coefficients at lags 1--12 are given in the second and forth columns of the output, with the lag length given in the first column.

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. 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 9 through 12) and the first two partial autocorrelation coefficients (then nine, 11 and 12) are significant under this rule. Since the first acf coefficient is highly significant, the joint test statistic presented in column 3 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.

[26] More information on the function **acf** can be viewed on-line from https://www.Statsmodels.org/dev/generated/Statsmodels.tsa.stattools.acf.html.

[27] We drop the first value of both the acf and pacf series by **acf[1:]** and pacf[1:] because the function starts to return coefficients at lag 0.

Using information criteria to decide on model orders

The formulae given in Brooks (2019) for Akaike's and Schwarz's Information Criteria are

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

$$SBIC = \ln(\hat{\sigma}^2) + \frac{k}{T}(\ln\ T) \tag{4}$$

where, $\hat{\sigma}^2$ is the estimator of the variance of the 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.

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.

Unlike other software, which may require users to separately estimate each of the models and to note down the value of the information criteria in each case, in Python, this can be easily done in one stage by employing the function **arma_order_select_ic** and it then generates a 5-by-5 DataFrame (if we specify the maximum order as five).

Initially, we will focus on one particular application to become familiar with this test – for example, an ARMA(1,1). To perform the ARMA regression, we call the built-in function **ARIMA** from the **statsmodels.tsa.api**. In the input bracket, we type the series **data['dhp']** and the order of the model, which would be **order=(1,0,1)** in this case. Note that there are three parameters in which we can type in the number of either the autoregressive order (p), the integrated (difference) order (d) or the moving-average order (q). As we want to start with estimating an **ARMA(1,1)** model, i.e., a model of **autoregressive order 1** and **moving-average order 1**, we specify this as the argument **order=(1,0,1)**.

We press **SHIFT** and **ENTER**, and Python generates the following estimation output.[28]

```
In [4]: res = smt.ARIMA(data['dhp'], order=(1,0,1)).fit()
        print(res.summary())
```

```
                              ARMA Model Results
==============================================================================
Dep. Variable:                    dhp   No. Observations:                  326
Model:                     ARMA(1, 1)   Log Likelihood                -462.710
Method:                       css-mle   S.D. of innovations              1.000
Date:                Mon, 20 Aug 2018   AIC                            933.420
Time:                        13:42:46   BIC                            948.567
Sample:                    02-01-1991   HQIC                           939.465
                         - 03-01-2018
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.4286      0.141      3.031      0.003       0.151       0.706
ar.L1.dhp      0.8224      0.060     13.781      0.000       0.705       0.939
ma.L1.dhp     -0.5417      0.088     -6.179      0.000      -0.714      -0.370
```

[28] Note that the Outer Product Gradient Standard Error(OPG SE) might be presented by other software such as STATA or E-views. This different treatment of the standard errors will cause different t-statistics compared to those from Python.

```
                                    Roots
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1            1.2160           +0.0000j            1.2160            0.0000
MA.1            1.8460           +0.0000j            1.8460            0.0000
-----------------------------------------------------------------------------
```

In theory, the output table would be discussed in a similar fashion 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 can simply use the built-in method **aic** from the **res** instance. It can be seen that the AIC has a value of 933.42 and the BIC a value of 948.57. 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 competing ARMA models and then to select the model with the lowest information criterion.

```
In [5]: print(res.aic)
        print(res.bic)

933.4199014912799
948.567491017
```

To check that the process implied by the model is stationary and invertible, it is useful to look at the inverses of the AR and MA roots of the characteristic equation. If the inverse roots of the AR polynomial all lie inside the unit circle, the process is stationary, invertible, and has an infinite-order moving-average (MA) representation. We can test this by selecting the sub-module **smt.ArmaProcess.from_estimation** and apply its built-in method **isinvertible** and **isstationary** individually. In the input bracket of the function, we enter the instance **res**. From the test output we see that the boolean results of both the AR and MA parts are true. Thus the conditions of stationarity and invertibility, respectively, are met.

```
In [6]: smt.ArmaProcess.from_estimation(res).isinvertible

Out[6]: True

In [7]: smt.ArmaProcess.from_estimation(res).isstationary

Out[7]: True
```

We can now turn to access the information criteria of all ARMA models, i.e., from ARMA(0,0) to ARMA(5,5). To obtain the table of all values of information criteria simultaneously, we select the function **arma_order_select_ic** from the **statsmodels.tsa.stattools** sub-module.

So which model actually minimises the two information criteria? In this case, the criteria choose different models: *AIC* selects an ARMA(5,4), while *SBIC* selects the smaller ARMA(2,0) model. These

chosen models can either be output by the built-in method **aic_min_order** and **bic_min_order** of the regression instance **res1** or visually examined the 5-by-5 information criteria DataFrame. Concerning the different model orders chosen, it is worth mentioning that 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.

```
In [8]: res1 = smt.arma_order_select_ic(data['dhp'],\
                                        max_ar=5, max_ma=5, ic=['aic', 'bic'],\
                                        fit_kw={'method':'css-mle',
                                                'solver':'bfgs'})
```

```
In [9]: print('AIC')
        print(res1.aic)
        print('SBIC')
        print(res1.bic)
```

```
AIC
             0           1           2           3           4           5
0  1001.263704  977.402015  935.508251  931.454458  930.294945  929.950512
1   958.791442  933.419902  925.648740  923.788834  924.499886  926.106244
2   922.460039  924.408036  926.242696  926.408807  925.977423  928.000364
3   924.401584  926.434415  928.183411  925.957411  925.429584  923.845552
4   926.261040  928.258696  928.377708  930.371143  931.083906  930.382511
5   928.245355  927.641964  927.246082  929.121258  917.127772  926.135325
SBIC
             0           1           2           3           4           5
0  1008.837498  988.762707  950.655841  950.388945  953.016330  956.458793
1   970.152134  948.567491  944.583227  946.510218  951.008168  956.401423
2   937.607629  943.342523  948.964081  952.917089  956.272602  962.082440
3   943.336070  949.155800  954.691693  956.252590  959.511661  961.714526
4   948.982425  954.766978  958.672887  964.453219  968.952880  972.038382
5   954.753636  957.937143  961.328159  966.990232  958.783644  971.578094
```

```
In [10]: print(res1.aic_min_order)
         print(res1.bic_min_order)
```

```
(5, 4)
(2, 0)
```

12 Forecasting using ARMA models

Reading: Brooks (2019, section 6.8)

Suppose that a AR(2) model selected for the house price percentage changes series were estimated using observations February 1991–December 2015, leaving 27 remaining observations for which to construct forecasts for and to test forecast accuracy (for the period January 2016–May 2018).

Let us first estimate the ARMA(2,0) model for the time period 1991-02-01 - 2015-12-01. As we only want to estimate the model over a subperiod of the data, we need to slice the **data**. To do so, we type the beginning and the end of the index values separated by a colon and assign the subsample to a new variable **data_insample** (see the command in In [2]). Next, we type the command **data_insample.tail()** and the last five rows of the DataFrame should appear in the window below. As you can see, the new variable ends in December 2015.

```
In [1]: import pickle
        import statsmodels.tsa.api as smt
        import NumPy as np
        from sklearn.metrics import mean_squared_error
        from math import sqrt

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'UKHP.pickle', 'rb') as handle:
            data = pickle.load(handle)

In [2]: data_insample = data['1991-02-01':'2015-12-01']
        data_insample.tail()

Out[2]:             Average House Price       dhp
        Month
        2015-08-01        195279.056782 -0.174970
        2015-09-01        195585.011303  0.156676
        2015-10-01        196807.127979  0.624852
        2015-11-01        196305.114697 -0.255079
        2015-12-01        196999.272687  0.353612
```

To perform the ARMA(2,0) model for the subsample, we repeat the steps described in the previous section. Specifically, we import the function **ARIMA** and feed the parameters: 'dhp' series and lag 2 for the autoregressive order (p). Next, the **model** instance is followed by the function **fit** to generate the regression results instance **res**. A summary table will appear in the window below by printing **res.summary()**.

```
In [3]: model = smt.ARIMA(data_insample['dhp'], order=(2,0,0))
        res = model.fit()
        print(res.summary())

                              ARMA Model Results
==============================================================================
Dep. Variable:                    dhp   No. Observations:                  299
Model:                 ARMA(2, 0)   Log Likelihood                -427.952
Method:                    css-mle   S.D. of innovations              1.012
```

```
Date:                Wed, 08 Aug 2018   AIC                            863.904
Time:                        19:44:34   BIC                            878.706
Sample:                  02-01-1991   HQIC                           869.828
                       - 12-01-2015
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.4417      0.137      3.225      0.001       0.173       0.710
ar.L1.dhp      0.2353      0.054      4.337      0.000       0.129       0.342
ar.L2.dhp      0.3406      0.054      6.259      0.000       0.234       0.447
                                    Roots
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1            1.4026           +0.0000j            1.4026            0.0000
AR.2           -2.0935           +0.0000j            2.0935            0.5000
-----------------------------------------------------------------------------
```

Now that we have fitted the model, we can produce the forecasts for the period 2016-01-01 - 2018-03-01. There are two methods available in Python for constructing forecasts: dynamic and static. The option **Dynamic** calculates 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.

We start with generating static forecasts. These forecasts can be generated by calling the built-in function **plot_predict** from the **res** instance. In the input bracket, we specify the prediction sample range: '2016-01-01' to '2018-03-01' and define the **dynamic** argument as **False**, which means implementing static forecasts. Once set up, you can press **SHIFT** and **ENTER** and the graph will be produced (Figure 24).

Likewise, we can create the dynamic forecasts in a similar way. This can be done very easily since we only need to change the argument **dynamic=False** to **dynamic=True**.

Let us have a closer look at the graph (Figure 25). 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.

```
In [4]: model = smt.ARIMA(data['dhp'], order=(2,0,0))
        res = model.fit()
        res.plot_predict('2016-01-01','2018-03-01',dynamic=False)
```

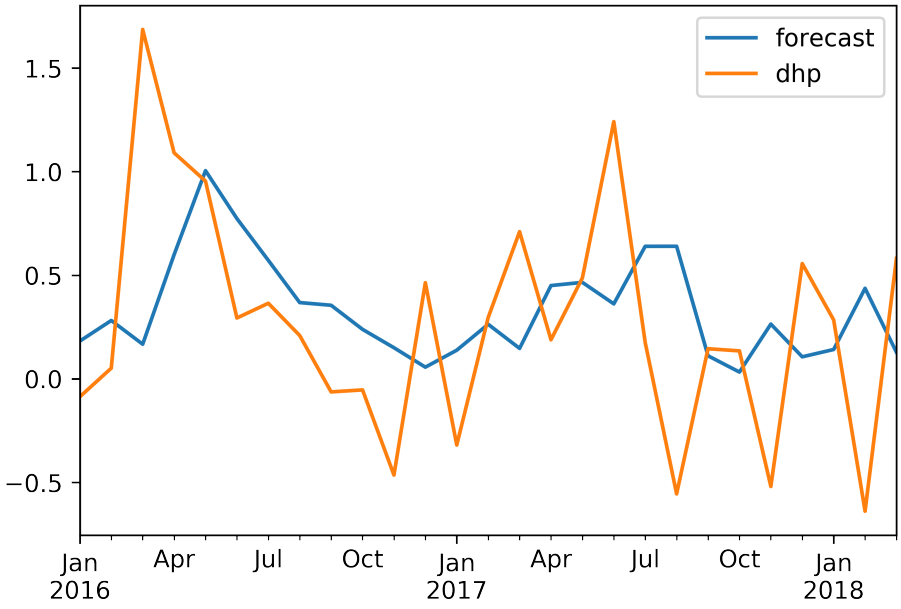


Figure 24: Graph Comparing the Static Forecasts with the Actual Series

```
In [5]: res.plot_predict('2016-01-01','2018-03-01',dynamic=True)
```

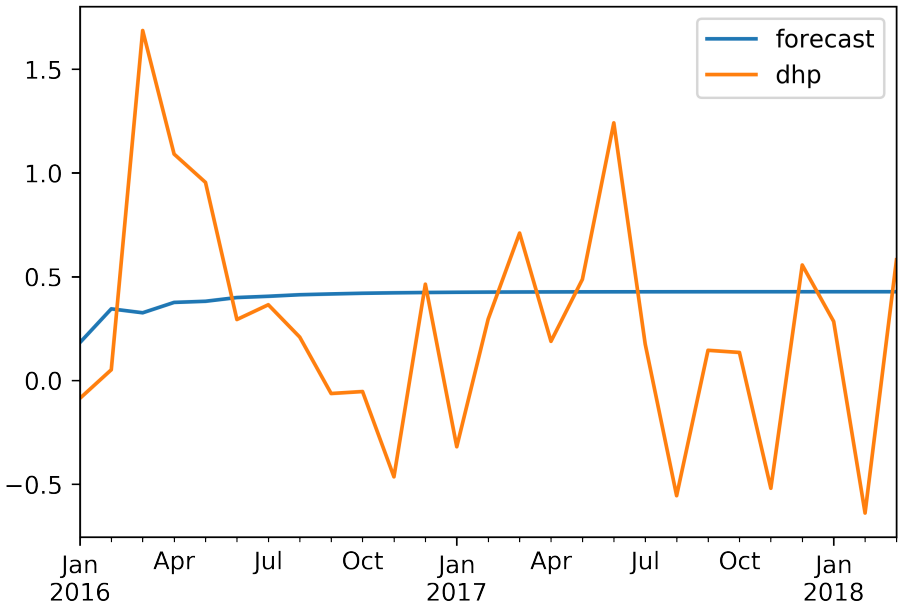


Figure 25: Graph Comparing the Dynamic Forecasts with the Actual Series

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.

In Python, RMSE can be easily calculated by some third-party Python packages. However, it is recommended for users to start from scratch given the formula.[29] To do so, we first define a custom function **rmse**. Specifically, we take the difference between each pair of the forecast and actual observations, then compute the average value of the squared differences and return the squared root of it. Next, we generate two new variables which are used to store the predicted and actual out-of-the-sample series, (**pred** and **data_outsample**). After that, we implement the function and print the statistic. The result will appear in the following window (In [6]). Alternatively, there is a built-in function available from some third-party packages. For example, the package **sklearn** contains the function **mean_squared_error**, which can compute these statistics but the module would need to be manually installed. With regard to the result, the **sklearn** module gives the same statistic as the one calculated by the custom function.

```
In [6]: def rmse(pred, target):
            return np.sqrt(((pred - target) ** 2).mean())

        data_outsample = data['2016-01-01':'2018-03-01']
        pred = res.predict('2016-01-01','2018-03-01',dynamic=False)

        stats1 = rmse(pred, data_outsample['dhp'])
        print('root mean squared error1: {}'.format(stats1) )

        stats2 = sqrt(mean_squared_error(data_outsample['dhp'], pred))
        print('root mean squared error2: {}'.format(stats2) )

root mean squared error1: 0.5776465575277504
root mean squared error2: 0.5776465575277503
```

[29] You can find the formulae to generate the forecast error statistics in chapter 6.11.8 of Brooks (2019).

13 Estimating exponential smoothing models

Reading: Brooks (2019, section 6.9)

Python allows us to estimate exponential smoothing models as well. To do so, we first re-load the 'UKHP.pickle' workfile and generate two subsample variables: **data_insample** and **data_outsample**. Next, we call the function **SimpleExpSmoothing** from the **statsmodels.tsa.api** and create a regression instance **model** based on the in-sample series 'dhp'. After generating the regression instance **res** by **fit**, we finally obtain the predicted values of the exponential smoothing model by the command **res.forecast(len(data_outsample))**. Note that there are only 27 observations for out-of-sample forecasting, and thus we tell Python this parameter by using the command **len(data_outsample)**.

```
In [1]: import pickle
        import statsmodels.tsa.api as smt
        import matplotlib.pyplot as plt
        import NumPy as np
        from math import sqrt

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'UKHP.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data_insample = data['1991-02-01':'2015-12-01']
        data_outsample = data['2016-01-01':'2018-03-01']

        model = smt.ExponentialSmoothing(data_insample['dhp'])
        res = model.fit()
        pred = res.predict(start='1991-02-01')
```

The result can be visualised in the NoteBook. As usual, we create a figure object and plot three separate variables: in-the-sample 'dhp', out-of-sample 'dhp' and forecasts (the detail of the commands is listed in the code cell [2]).

Let us examine the output graph in Figure 26. It is clearly illustrated that the in-sample 'dhp' is presented in blue while the out-of-sample values are shown in orange. On the rightmost side of the graph, there is a green line which represents the simple exponential smoothing forecasts.

```
In [2]: plt.figure(1)
        plt.plot(data_insample['dhp'], label='In-the-sample Data')
        plt.plot(data_outsample['dhp'], label='Out-of-the-sample Data')
        plt.plot(pred, label='Simple Exponential Smoothing')
        plt.legend()
        plt.show()
```

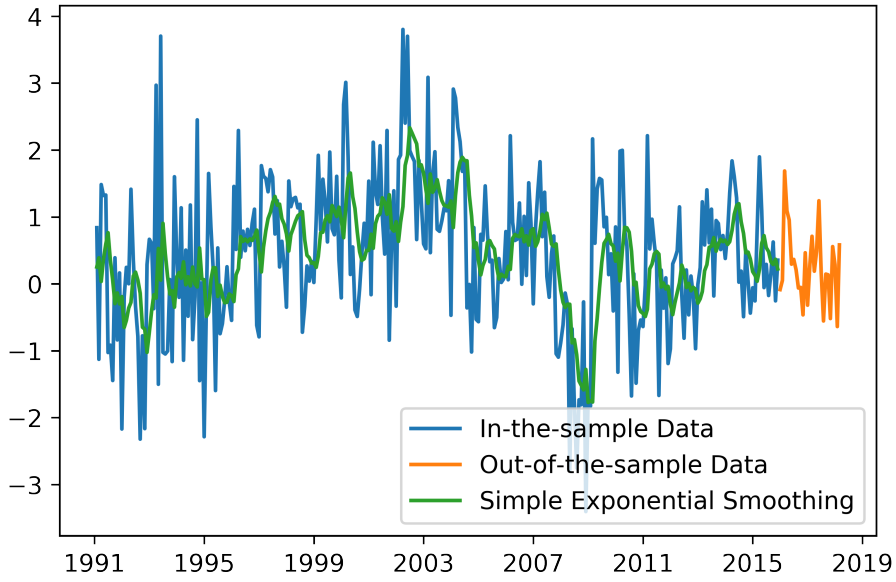


Figure 26: In-sample, Out-of-sample and Simple Exponential Smoothing

Additionally, it is preferable to compute the sum-of-squared residuals (RSS) for the in-sample estimation period and the root mean squared error (RMSE) for the 27 forecasts. Since the step of calculating the RMSE has been discussed in the previous section, it is easy to get this statistic by re-using those commands. Moreover, the RSS can be accessed by the command **res.sse**.

```
In [3]: def rmse(pred, target):
            return np.sqrt(((pred - target) ** 2).mean())

        stats = rmse(pred,data_insample['dhp'])
        print('Optimal smoothing coefficient: {}'.format(res.params['smoothing_level']))
        print('root mean squared error: {}'.format(stats) )
        print('sum-of-squared residuals: {}'.format(res.sse) )

Optimal smoothing coefficient: 0.23279238720990583
root mean squared error: 1.0586557811514028
sum-of-squared residuals: 335.1048668266209
```

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 employs the same macroeconomic data as used previously 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:

$$inflation_t = \alpha_0 + \alpha_1 returns_t + \alpha_2 dcredit_t + \alpha_3 dprod_t + \alpha_4 dmoney + u_{1t} \quad (5)$$

$$returns_t = \beta_0 + \beta_1 dprod_t + \beta_2 dspread_t + \beta_3 inflation_t + \beta_4 rterm_t + u_{2t} \quad (6)$$

where 'returns' are stock returns.

It is evident that there is feedback between the two equations since the *inflation* variable appears in the *stock returns* equation and vice versa. Are the equations identified? Since there are two equations, each will be identified if one variable is missing from that equation. Equation (5), the inflation equation, omits two variables. It does not contain the default spread or the term spread, and so is over-identified. Equation (6), the stock returns equation, omits two variables as well -- the consumer credit and money supply variables, and so it over-identified too. 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:

$$inflation = f(constant, dprod, dspread, rterm, dcredit, qrev, dmoney) \quad (7)$$

$$returns = g(constant, dprod, dspread, rterm, dcredit, qrev, dmoney) \quad (8)$$

For this example we will be using the 'macro.pickle' file. To perform a 2SLS regression, we need to import a third-party package **linearmodels**.[30]

There are a variety of built-in functions from this module for advanced econometric applications. To fit our purpose, we can directly import the function **IV2SLS**.

```
In [1]: import pickle
        from linearmodels import IV2SLS
        import statsmodels.api as sm
        import Pandas as pd

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'

        with open(abspath + 'macro.pickle', 'rb') as handle:
            data = pickle.load(handle)
```

[30] To install the linearmodels package, you need to press **START** and search for the 'Anaconda Prompt'. Once the window opens, type the command **pip install linearmodels** and hit **ENTER**.

The steps for constructing regression specifications remain similar, albeit via a different library. The first step of these would be adding a constant term to the data by the function **add_constant** from Statsmodels. Subsequently, the 2SLS regression model instance is created by performing the function **IV2SLS**. In its brackets, we specify four parameters: the dependent variable, the exogenous variable, the endogenous variable and the instruments. Specifically, the first parameter **dependent** is defined as the series 'inflation'. **exog** are **const dprod dcredit dmoney**, while the variable **rsandp** is set as **endog**. Last but not least, the list of **Instruments** comprises the variables 'rterm' and 'dspread'.

The **res_2sls** regression result instance is then generated by the function **fit**. However, in this case, we type the argument **cov_type='unadjusted'** for the function.

```
In [2]: # 2SLS, specification 1
        data = sm.add_constant(data)
        ivmod = IV2SLS(dependent = data.inflation,\
                       exog = data[['const','dprod','dcredit','dmoney']],\
                       endog = data.rsandp,\
                       instruments = data[['rterm','dspread']])
        res_2sls1 = ivmod.fit(cov_type='unadjusted')
        print(res_2sls1)
```

```
                          IV-2SLS Estimation Summary
==============================================================================
Dep. Variable:              inflation   R-squared:                      -1.8273
Estimator:                    IV-2SLS   Adj. R-squared:                 -1.8571
No. Observations:                 384   F-statistic:                     21.784
Date:                Fri, Nov 09 2018   P-value (F-stat)                 0.0002
Time:                        11:39:19   Distribution:                  chi2(4)
Cov. Estimator:            unadjusted

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
const          0.2129     0.0369     5.7777     0.0000      0.1407      0.2852
dprod          0.0309     0.0500     0.6172     0.5371     -0.0671      0.1289
dcredit       -0.0052     0.0019    -2.7214     0.0065     -0.0089     -0.0015
dmoney        -0.0028     0.0011    -2.6408     0.0083     -0.0049     -0.0007
rsandp         0.1037     0.0333     3.1092     0.0019      0.0383      0.1690
==============================================================================

Endogenous: rsandp
Instruments: rterm, dspread
Unadjusted Covariance (Homoskedastic)
Debiased: False
```

Similarly, the inputs for the 'rsandp' equation would be specified as in the follow code cell and the output for the returns equation is shown below.

```
In [3]: # 2SLS, specification 2
        ivmod = IV2SLS(dependent = data.rsandp,\
```

```
                    exog = data[['const','dprod','dcredit','dmoney']],\
                    endog = data.inflation,\
                    instruments = data[['rterm','dspread']])
res_2sls2 = ivmod.fit(cov_type='unadjusted')
print(res_2sls2)
```

```
                          IV-2SLS Estimation Summary
==============================================================================
Dep. Variable:                  rsandp   R-squared:                     -0.1795
Estimator:                     IV-2SLS   Adj. R-squared:                -0.1920
No. Observations:                  384   F-statistic:                    8.4611
Date:                Fri, Nov 09 2018   P-value (F-stat)                0.0761
Time:                        11:39:19   Distribution:                  chi2(4)
Cov. Estimator:            unadjusted

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
const         -1.1624     0.6697    -1.7357     0.0826     -2.4750      0.1502
dprod         -0.2366     0.4376    -0.5406     0.5888     -1.0942      0.6211
dcredit        0.0368     0.0186     1.9851     0.0471      0.0005      0.0732
dmoney         0.0185     0.0108     1.7087     0.0875     -0.0027      0.0397
inflation      6.3039     2.2728     2.7736     0.0055      1.8493      10.759
==============================================================================

Endogenous: inflation
Instruments: rterm, dspread
Unadjusted Covariance (Homoskedastic)
Debiased: False
```

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 also has a positive effect on the stock returns, albeit less significantly so.

15 The Generalised method of moments for instrumental variables

Reading: Brooks (2019, sections 7.8 and 14.4)

Apart from 2SLS, there are other ways to address the endogeneity issue in a system of equations. Following the previous section using inflation and stock returns, we apply the same "macro.pickle" Python workfile to explore a different technique: the Generalised Method of Moments (GMM). First, recall the estimated models as follows:

$$inflation_t = \alpha_0 + \alpha_1 returns_t + \alpha_2 dcredit_t + \alpha_3 dprod_t + \alpha_4 dmoney + u_{1t} \quad (9)$$

$$returns_t = \beta_0 + \beta_1 dprod_t + \beta_2 dspread_t + \beta_3 inflation_t + \beta_4 rterm_t + u_{2t} \quad (10)$$

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

Clearly, there is feedback between the two equations since the *inflation* variable appears in the *stock returns* equation and vice versa. Therefore, the GMM is employed with a list of instruments to be tested. To perform the GMM, we again use the third-party package **linearmodels**. We can directly import the function **IVGMM** this time.

```
In [1]: import pickle
        from linearmodels.system import IVSystemGMM
        from linearmodels.iv import IVGMM
        import statsmodels.api as sm
        import Pandas as pd

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'

        with open(abspath + 'macro.pickle', 'rb') as handle:
            data = pickle.load(handle)
```

The steps for constructing regression specifications are the same as the previous ones, albeit via a different regression function. The first step is writing the regression formula. Within the GMM setting, the regression model is the dependent variable followed by the exogenous variables with endogenous variables and instruments added afterwards in squared brackets. Specifically, the formula statement is as follows:

'inflation ~ 1 + dprod + dcredit + dmoney + [rsandp ~ rterm + dspread]'

where inflation is the dependent variable; 1 is the constant term; dprod, dcredit, and dmoney are exogenous variables; rsandp is the endogenous variable; rterm and dspread are instruments. Subsequently, the GMM regression model instance is created by performing the function **IVGMM.from_formula**. In its brackets, we input the formula specified, data and weighting scheme. Next, the covariance type is set as robust in the **fit**. Executing the cell will lead the regression results to be displayed in the output window.

```
In [2]: # GMM, specification 1
        formula = 'inflation ~ 1 + dprod + dcredit + dmoney + [rsandp ~ rterm + dspread]'
        mod = IVGMM.from_formula(formula, data, weight_type='unadjusted')
        res1 = mod.fit(cov_type='robust')
        print(res1.summary)
```

```
                          IV-GMM Estimation Summary
==============================================================================
Dep. Variable:              inflation   R-squared:                      -1.8273
Estimator:                     IV-GMM   Adj. R-squared:                 -1.8571
No. Observations:                 384   F-statistic:                     20.058
Date:                Sun, Nov 11 2018   P-value (F-stat)                 0.0005
Time:                        22:16:12   Distribution:                   chi2(4)
Cov. Estimator:                robust

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
Intercept      0.2129     0.0422     5.0416     0.0000      0.1302      0.2957
dprod          0.0309     0.0699     0.4413     0.6590     -0.1062      0.1679
dcredit       -0.0052     0.0017    -2.9732     0.0029     -0.0086     -0.0018
dmoney        -0.0028     0.0011    -2.5944     0.0095     -0.0049     -0.0007
rsandp         0.1037     0.0419     2.4768     0.0133      0.0216      0.1857
==============================================================================

Endogenous: rsandp
Instruments: rterm, dspread
GMM Covariance
Debiased: False
Robust (Heteroskedastic)
```

Similarly, the second specification for the 'rsandp' equation would be written as in the following code cell and the output for the returns equation is shown below.

```
In [3]: # GMM, specification 2
        formula = 'rsandp ~ 1 + dprod + dcredit + dmoney + [inflation ~ rterm + dspread]'
        mod = IVGMM.from_formula(formula, data, weight_type='unadjusted')
        res2 = mod.fit(cov_type='robust')
        print(res2.summary)
```

```
                          IV-GMM Estimation Summary
==============================================================================
Dep. Variable:                  rsandp   R-squared:                      -0.1795
Estimator:                     IV-GMM   Adj. R-squared:                 -0.1920
No. Observations:                 384   F-statistic:                     5.6843
Date:                Sun, Nov 11 2018   P-value (F-stat)                 0.2240
Time:                        22:16:12   Distribution:                   chi2(4)
Cov. Estimator:                robust

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
```

```
Intercept     -1.1624     0.8919    -1.3033     0.1925     -2.9105      0.5857
dprod         -0.2366     0.6369    -0.3714     0.7103     -1.4849      1.0117
dcredit        0.0368     0.0185     1.9929     0.0463      0.0006      0.0731
dmoney         0.0185     0.0104     1.7849     0.0743     -0.0018      0.0388
inflation      6.3039     3.1875     1.9777     0.0480      0.0565      12.551
==============================================================================

Endogenous: inflation
Instruments: rterm, dspread
GMM Covariance
Debiased: False
Robust (Heteroskedastic)
```

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 also has a positive effect on the stock returns, albeit less significantly so.

16 VAR estimation

Reading: Brooks (2019, section 7.10)

In this section, a VAR 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 the NoteBook. Next, we construct a set of continuously compounded percentage returns called **'reur', 'rgbp' and 'rjpy'** using a custom function **LogDiff**. Moreover, these new variables are saved in a workfile **currencies.pickle** for the future usage.

```
In [1]: import Pandas as pd
        import NumPy as np
        import statsmodels.tsa.api as smt
        import pickle

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'

        data = pd.read_excel(abspath + 'currencies.xls',index_col=[0])

        def LogDiff(x):
            x_diff = 100*np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data = pd.DataFrame({'reur':LogDiff(data['EUR']),
                             'rgbp':LogDiff(data['GBP']),
                             'rjpy':LogDiff(data['JPY'])})

        with open(abspath + 'currencies.pickle', 'wb') as handle:
            pickle.dump(data, handle)
```

VAR estimation in Python can be accomplished by importing the function **VAR** from the library **statsmodels.tsa.api**. The VAR specification appears as in the code cell [2]. We define the dependent variables to be 'reur', 'rgbp' and 'rjpy'. Next we need to specify the number of lags to be included for each of these variables. In this case, the maximum number of lags is two, i.e., the first lag and the second lag. Let us write the argument **maxlags=2** for the regression instance and estimate this VAR(2) model. The regression output appears as below.

```
In [2]: # VAR
        model = smt.VAR(data)
        res = model.fit(maxlags=2)
        print(res.summary())

  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Mon, 20, Aug, 2018
```

```
Time:                      15:35:25
--------------------------------------------------------------------
No. of Equations:         3.00000    BIC:                   -5.41698
Nobs:                     7139.00    HQIC:                  -5.43024
Log likelihood:          -10960.3    FPE:                  0.00435167
AIC:                     -5.43720    Det(Omega_mle):       0.00433889
--------------------------------------------------------------------
```

Results for equation reur

	coefficient	std. error	t-stat	prob
const	0.000137	0.005444	0.025	0.980
L1.reur	0.147497	0.015678	9.408	0.000
L1.rgbp	-0.018356	0.017037	-1.077	0.281
L1.rjpy	-0.007098	0.012120	-0.586	0.558
L2.reur	-0.011808	0.015663	-0.754	0.451
L2.rgbp	0.006623	0.017032	0.389	0.697
L2.rjpy	-0.005427	0.012120	-0.448	0.654

Results for equation rgbp

	coefficient	std. error	t-stat	prob
const	0.002826	0.004882	0.579	0.563
L1.reur	-0.025271	0.014058	-1.798	0.072
L1.rgbp	0.221362	0.015277	14.490	0.000
L1.rjpy	-0.039016	0.010868	-3.590	0.000
L2.reur	0.046927	0.014045	3.341	0.001
L2.rgbp	-0.067794	0.015272	-4.439	0.000
L2.rjpy	0.003287	0.010868	0.302	0.762

Results for equation rjpy

	coefficient	std. error	t-stat	prob
const	-0.000413	0.005524	-0.075	0.940
L1.reur	0.041061	0.015908	2.581	0.010
L1.rgbp	-0.070846	0.017287	-4.098	0.000
L1.rjpy	0.132457	0.012298	10.771	0.000
L2.reur	-0.018892	0.015893	-1.189	0.235
L2.rgbp	0.024908	0.017282	1.441	0.150
L2.rjpy	0.014957	0.012298	1.216	0.224

Correlation matrix of residuals

	reur	rgbp	rjpy
reur	1.000000	0.634447	0.270764

```
rgbp    0.634447  1.000000  0.164311
rjpy    0.270764  0.164311  1.000000
```

At the top of the table, we find information for the model as a whole, including values of the information criteria, while further down we find coefficient estimates and goodness-of-fit measures for each of the equations separately. Each regression equation is separated by a horizontal line.

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. This can be achieved in a variety of ways, but one of the easiest is to employ a multivariate information criterion. In Python, this can be done by calling the built-in function **select_order** based on the regression model instance. In the specification brackets of the function, we specify the maximum number of lags to entertain including in the model, and for this example, we arbitrarily enter **10**. By executing the code cell, we should be able to observe the following output.

```
In [3]: res = model.select_order(maxlags=10)
        print(res.summary())
```

```
 VAR Order Selection (* highlights the minimums)
==================================================
       AIC         BIC         FPE         HQIC
--------------------------------------------------
0       -5.346      -5.343    0.004769      -5.345
1       -5.433     -5.422*    0.004368      -5.429
2       -5.437      -5.417    0.004351     -5.430*
3       -5.438      -5.409    0.004350      -5.428
4      -5.439*      -5.401   0.004344*      -5.426
5       -5.438      -5.392    0.004346      -5.423
6       -5.437      -5.382    0.004351      -5.418
7       -5.437      -5.373    0.004353      -5.415
8       -5.436      -5.364    0.004358      -5.411
9       -5.435      -5.354    0.004360      -5.407
10      -5.434      -5.344    0.004367      -5.403
--------------------------------------------------
```

Python presents the values of various information criteria and other methods for determining the lag order. In this case, the Akaike (AIC) and Akaike's Final Prediction Error Criterion (FPE) both select a lag length of four as optimal, while Schwarz's (SBIC) criterion chooses a VAR(1) and the Hannan-Quinn (HQIC) criteria selects a VAR(2). 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?

Next, we run a Granger causality test. We call the **statsmodels.tsa.api** function **VAR** again and construct a VAR regression instance. Next, we run the built-in function **test_causality** and specify the parameters as follows: causing variable 'rgbp', caused variable 'reur', performing test options 'wald' and significance level for computing critical values, 0.05. It is unfortunate that the Granger causality can only be tested between two variables in Python since we want to run the test among

all variables. However, this can be done by separately estimating each of the pairwise combinations and noting down the statistics in each case (Table 1).[31]

```
In [4]: model = smt.VAR(data)
        res = model.fit(maxlags=2)

        #------------------------------------------------
        # Equation reur, Excluded rgbp
        resCausality = res.test_causality(causing=['rgbp'],
                                  caused=['reur'],
                                  kind='wald',signif=0.05 )
```

[31] In the code cell, we only demonstrate the command for the equation reur due to limited space. Users can modify it for other equations.

Table 1: Granger Causality Wald tests

Equation	Excluded	chi2	Critical value	p-value	df
reur	rgbp	1.186	5.991	0.553	2
reur	rjpy	0.6260	5.991	0.731	2
reur	All	1.764	9.488	0.779	4
rgbp	reur	12.88	5.991	0.002	2
rgbp	rjpy	12.92	5.991	0.002	2
rgbp	All	28.99	9.488	0.000	4
rjpy	reur	7.320	5.991	0.026	2
rjpy	rgbp	17.12	5.991	0.000	2
rjpy	All	17.38	9.488	0.002	4

The results show only modest evidence of lead-lag interactions between the series. Since we have estimated a tri-variate VAR, three panels are displayed, with one for each dependent variable in the system. There is causality from EUR to GBP and from JPY to GBP that is significant at the 1% level. We also find significant causality at the 5% level from EUR to JPY and GBP to JPY, but no causality from any of the currencies to EUR. These results might be interpreted as suggesting that information is incorporated slightly more quickly in the pound-dollar rate and yen-dollar rates than into the euro-dollar rate.

It is preferable to visualise the impact of changes in one variable on the others at different horizons. One way to achieve this is to obtain the impulse responses for the estimated model. To do so, we first re-define the dependent variables (reur rgbp rjpy) and select a VAR model with one lag of each variable. This can be done by inputting the argument **maxlags=1**. We then specify that we want to generate a graph for the **irf**, that is, the built-in function from the VAR result instance. Finally, we need to select the number of periods over which we want to generate the IRFs. We arbitrarily select 20 and feed it to **irf**. Type the command **irf.plot()** and Python produces the impulse response graphs (Figure 27) as below.

```
In [5]: model = smt.VAR(data)
        res = model.fit(maxlags=1)

        # Impulse Response Analysis
        irf = res.irf(20)
        irf.plot()

   Out[5]:
```

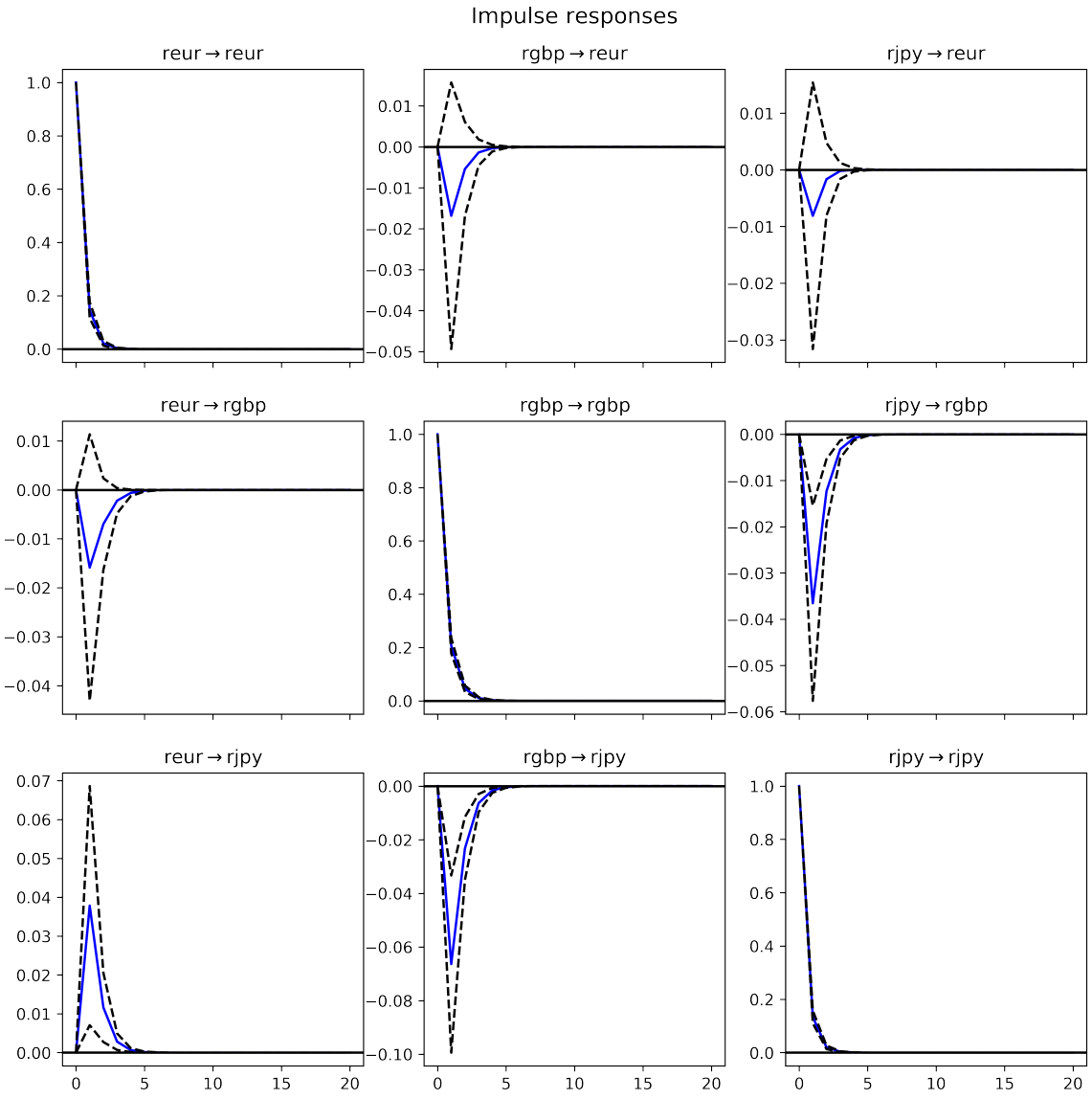


Figure 27: Impulse Responses

As one would expect given the parameter estimates and the Granger causality test results, 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.

Note that plots of the variance decompositions (also known as forecast error variance decompositions, or fevd in Python) can also be generated using the **fevd** function. Instead of plotting the IRFs by the **irf** function, we choose **fevd** – that is, the forecast-error variance decompositions. Bar charts for the variance decompositions would appear as follows (see Figure 28).

```
In [6]: # Forecast Error Variance Decomposition (FEVD)
```

```
fevd = res.fevd(20)
fevd.plot()
```

Out[6]:

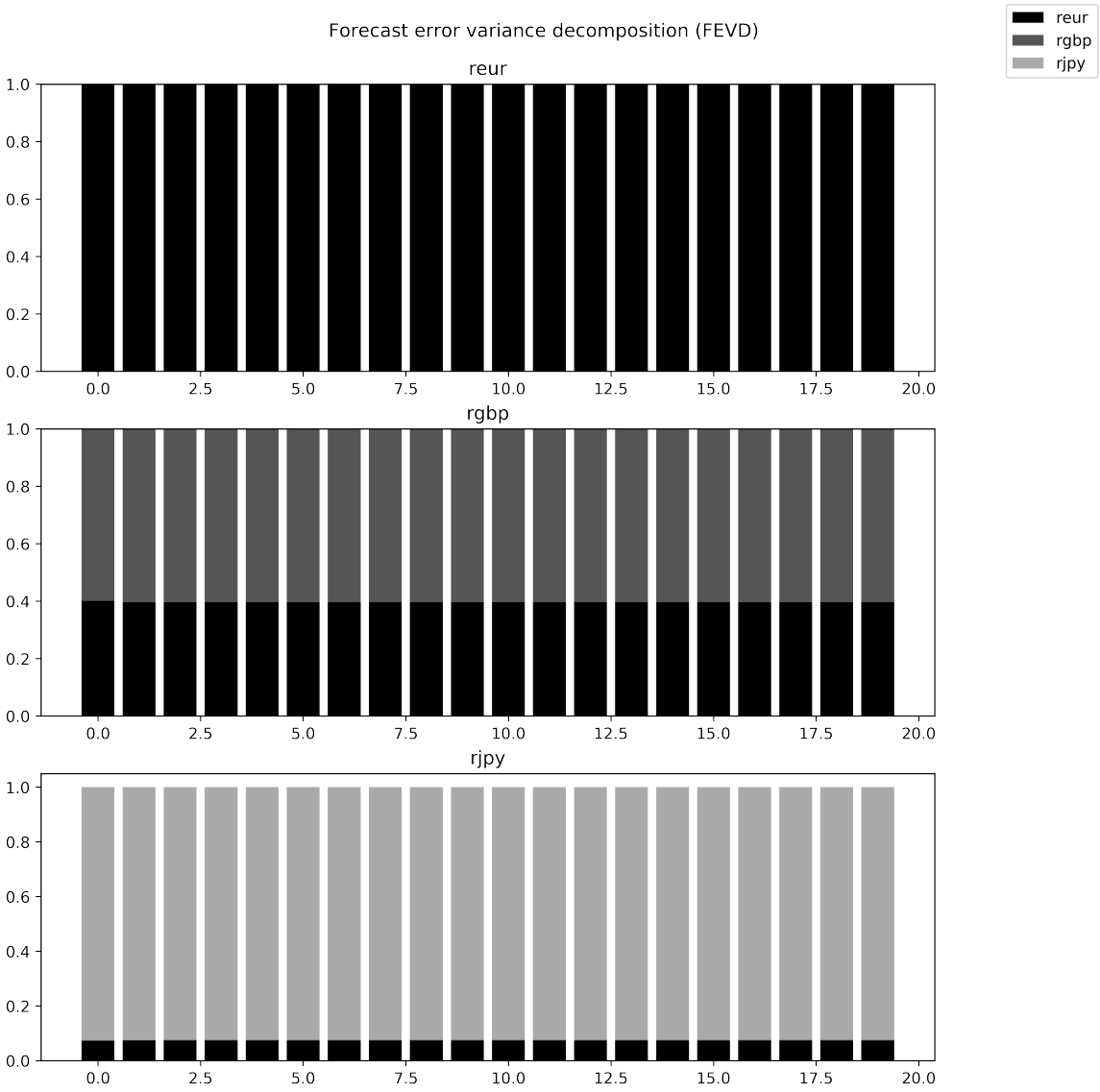


Figure 28: Variance Decompositions

To illustrate how to interpret the FEVDs, let us have a look at the effect that a shock to the euro rates has on the other two rates and on later values of the euro series 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 (dark black bar), for the pound, the euro series explains around 40% of the variation in returns (top 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. We first generate a new DataFrame **data1** with the reverse order of the columns to be used previously, which is **rjpy rgbp reur**. To inspect and compare the FEVDs for this ordering and the previous one, we can create graphs of the FEVDs by implementing the **VAR** regression and **fevd** function again. We can then compare the FEVDs of the reverse order (Figure 29) with those of the previous order.

```
In [7]: data1 = data[['rjpy','rgbp','reur']] # reverse the columns

        model = smt.VAR(data1)
        res = model.fit(maxlags=1)

        # Forecast Error Variance Decomposition (FEVD)
        fevd = res.fevd(20)
        fevd.plot()
```

Out[7]:

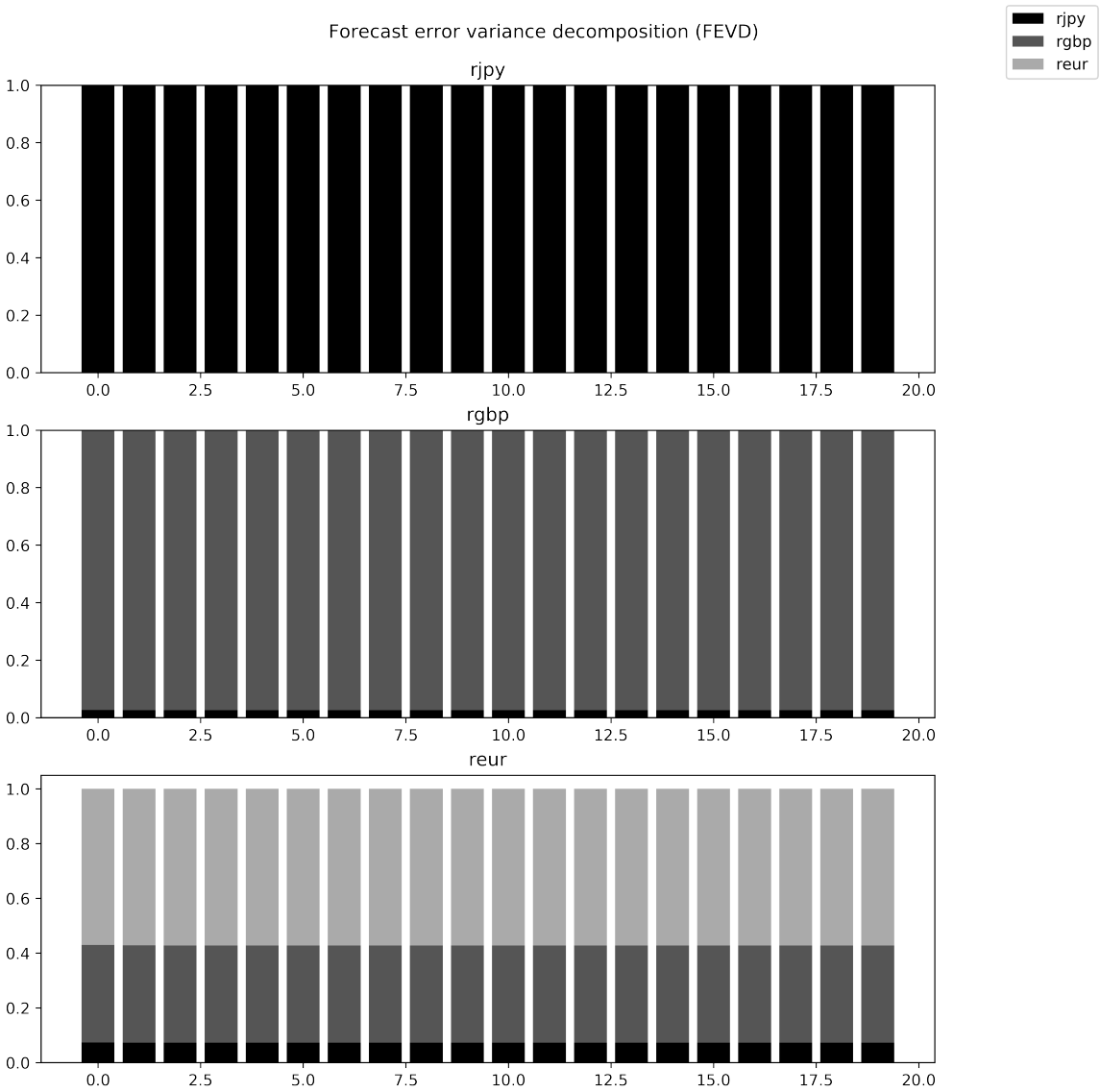


Figure 29: Variance Decompositions for Different Orderings

17 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 using Python. This example uses the same data on UK house prices as employed previously ('ukhp.pickle'). Assuming that the data have been loaded, the variables are defined and that we have dropped missing values as before, we want to conduct a unit root test on the **Average House Price** series. To start with, we choose a new third-party package, **arch**, rather than sticking to **statsmodels** because the former contains a greater number of unit root test tools.[32]

```
In [1]: import pickle
        from arch.unitroot import DFGLS, ADF, KPSS, PhillipsPerron

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'UKHP.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data['First Difference of HP'] = data['Average House Price'].diff()
        data = data.dropna()
        data.head()
```

```
Out[1]:             Average House Price       dhp  First Difference of HP
        Month
        1991-03-01         52892.861606 -1.128922             -603.937141
        1991-04-01         53677.435270  1.483326              784.573665
        1991-05-01         54385.726747  1.319533              708.291476
        1991-06-01         55107.375085  1.326908              721.648338
        1991-07-01         54541.121263 -1.027546             -566.253821
```

The first test we employ will be the Augmented Dickey-Fuller (ADF) unit-root test. Since these functions have been imported from the **arch.unitroot** sub-module, we can directly call the function **ADF** and input the parameters, which are the series 'Average House Price' and 10 Lagged differences, respectively. Next, we type the command **print(res.summary())** as usual and the following test statistics are reported in the output window below.

In the upper part of the output, we find the actual test statistics for the null hypothesis that the series 'Average House Price' has a unit root. Clearly, the test statistic (–0.467) is not more negative than the critical value, so the null hypothesis of a unit root in the house price series cannot be rejected.

```
In [2]: # test level
        res = ADF(data['Average House Price'], lags=10)
        print(res.summary())
```

```
   Augmented Dickey-Fuller Results
====================================
Test Statistic                 -0.467
P-value                         0.898
```

[32] To install this package, press **START** and search for the 'Anaconda Prompt'. Once the window opens, you need to type the command **pip install arch** and execute it by hitting **ENTER**.

```
Lags                                10
-------------------------------------

Trend: Constant
Critical Values: -3.45 (1%), -2.87 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

Now we repeat all of the above steps for the **First Difference of HP**. To do so, we call the **ADF** function again but instead of typing **data['Average House Price']**, we type **data['First Difference of HP']**. The output would appear as in the code cell below.

```
In [3]: res = ADF(data['First Difference of HP'], lags=10)
        print(res.summary())

   Augmented Dickey-Fuller Results
=====================================
Test Statistic                -3.307
P-value                        0.015
Lags                              10
-------------------------------------

Trend: Constant
Critical Values: -3.45 (1%), -2.87 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

We find that the null hypothesis of a unit root can be rejected for the differenced house price series at the 5% level.[33]

For completeness, we run a unit root test on the **dhp** series (levels, not differenced), which are the percentage changes rather than the absolute differences in prices. We should find that these are also stationary (at the 5% level for a lag length of 10).

```
In [4]: res = ADF(data['dhp'], lags=10)
        print(res.summary())

   Augmented Dickey-Fuller Results
=====================================
Test Statistic                -3.220
P-value                        0.019
Lags                              10
-------------------------------------

Trend: Constant
Critical Values: -3.45 (1%), -2.87 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

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

As mentioned above, the arch module presents a large number of options for unit root tests. We could, for example, include a trend or a drift term in the ADF regression. Alternatively, we could 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 in the arch library, we 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 (**DFGLS**). It can be accessed in Python via **arch.unitroot**. 'DFGLS' performs a modified Dickey–Fuller *t*-test for a unit root in which the series has been transformed by a generalized least-squares regression. Several empirical studies have shown that this test has significantly greater power than previous versions of the augmented Dickey–Fuller test. Another advantage of the 'DFGLS' test is that it does not require knowledge of the optimal lag length before running it but it performs the test for a series of models that include 1 to k lags.

In the **DFGLS**'s input, shown in the code cell below, we type **data['Average House Price']** and specify the highest lag order for the Dickey-Fuller GLS regressions to be 10. We then write the command **print(res.summary())** and Jupyter generates the following output.

```
In [5]: res = DFGLS(data['Average House Price'], max_lags=10)
        print(res.summary())

      Dickey-Fuller GLS Results
====================================
Test Statistic                  1.726
P-value                         0.979
Lags                                2
------------------------------------

Trend: Constant
Critical Values: -2.63 (1%), -2.01 (5%), -1.69 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

We see the optimal number of lags selected by the test output is two, albeit choosing only from one lag to ten. For two lags, we have a test statistic of 1.726, which is not more negative than the critical value, even at the 10% level.

Apart from these two tests – the **ADF** and **DFGLS** – one can also apply other unit root tests such as the KPSS, Phillips-Perron and so on, but conducting these is left as an exercise.

18 Cointegration tests and modelling cointegrated systems

Reading: Brooks (2019, sections 8.3 – 8.11)
In this section, we will test the S&P500 spot and futures series in the 'SandPhedge.xls' Excel workfile (that were discussed in section 3) for cointegration using Python.

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 generate two new variables, for the log of the spot series and the log of the futures series, and rename them **lspot** and **lfutures**, respectively.[34] Then we run the following OLS regression:

$$\textbf{lspot} \sim \textbf{lfuture}$$

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 residual series over time. As explained in previous sections, we can use the built-in method **resid** and **fittedvalues** from the **results** instance to generate series of the residuals and fitted values.

```
In [1]: import Pandas as pd
        import NumPy as np
        import statsmodels.formula.api as smf
        import matplotlib.pyplot as plt
        from arch.unitroot import DFGLS

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD' \
                        '/QMF Book/book Ran/data files new/Book4e_data/'
        data = pd.read_excel(abspath + 'SandPhedge.xls', index_col=0)

        data['lspot'] = data['Spot'].apply(lambda x : np.log(x))
        data['lfuture'] = data['Futures'].apply(lambda x : np.log(x))

        formula = 'lspot ~ lfuture'
        results = smf.ols(formula, data).fit()

        residuals = results.resid
        lspot_fit = results.fittedvalues
```

Next we generate a graph of the actual, fitted and residual series by calling the **matplotlib.pyplot** module. Note that we have created a second y-axis for the residuals as they are very small and we would not be able to observe their variation if they were plotted on the same scale as the actual and fitted values.[35] The plot should appear as follows (see Figure 30):

```
In [2]: fig = plt.figure(1)
        ax1 = fig.add_subplot(111)
        ax1.plot(lspot_fit, label='Linear Prediction')
```

[34] We use the command **data['lspot'] = data['Spot'].apply(lambda x : np.log(x))** to generate the series. Note that it is common to run a regression of the log of the spot price on the log of the futures rather than a regression in levels; the main reason for using logarithms is that the differences of the logs are returns, whereas this is not true for the levels.

[35] When using the **matplotlib.pyplot** to create the graph, you can generate an axis object by typing the command **ig.add_subplot(111)**. Next, you can create another axis object by applying the command **ax2 = plt.twinx()** when defining the plot for the residuals.

```
ax1.plot(data['lspot'], label='lspot')
ax1.set_xlabel('Date')
ax1.legend(loc=0)

ax2 = plt.twinx()
ax2.set_ylabel('Residuals')
ax2.plot(residuals, label='Residuals')
ax2.legend(loc=0)

plt.grid(True)
plt.show()
```

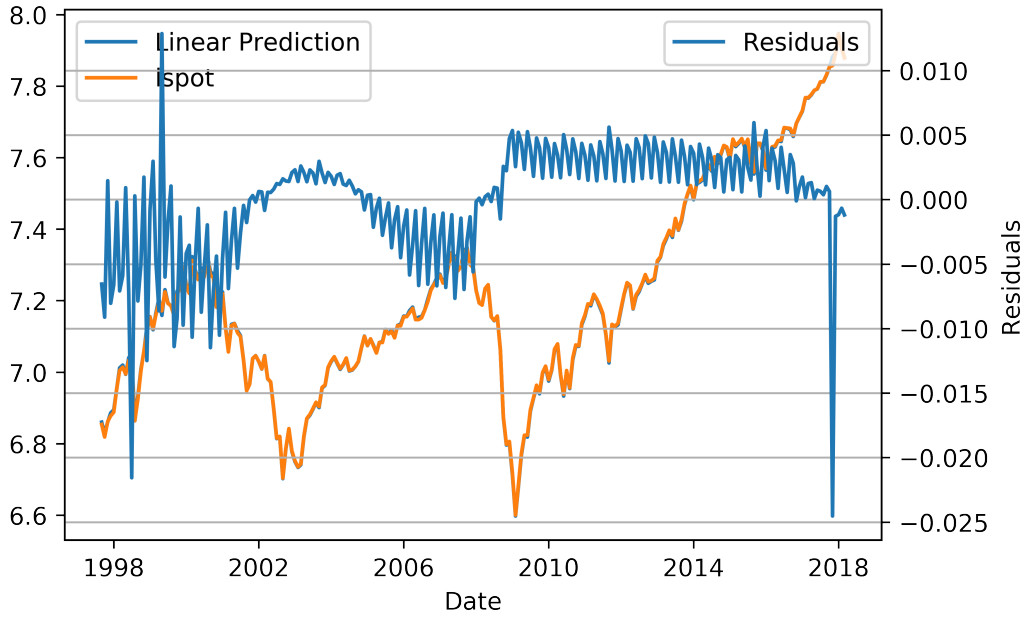


Figure 30: Actual, Fitted and Residual Plot

You will see a plot of the levels of the residuals (red line), which looks much more like a stationary series than the original spot series (the blue 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.

Let us now perform an ADF test on the residual series '**residuals**'. As we do not know the optimal lag length for the test, we use the **DFGLS** test and specify **12** max_lags as the highest lag order for DFGLS regressions. The output should appear as below.

The number of lags displayed at the bottom of the test output suggests an optimal lag length of 9, since we focus on the Akaike information criterion (AIC) by default. For two lags, we have a test statistic of (–0.925), 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.

```
In [3]: res = DFGLS(residuals, max_lags=12)
        print(res.summary())

      Dickey-Fuller GLS Results
====================================
Test Statistic                 -0.925
P-value                         0.324
Lags                                9
------------------------------------

Trend: Constant
Critical Values: -2.65 (1%), -2.03 (5%), -1.71 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

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:

$$\textbf{'rspot} \sim \textbf{rfuture + lresid'} \tag{11}$$

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 in any case.

```
In [4]: # Error Correction Model
        # specification 1: rspot rfutures L.resid
        def LogDiff(x):
            x_diff = 100*np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data['rspot'] = LogDiff(data['Spot'])
        data['rfuture'] = LogDiff(data['Futures'])
        data['lresid'] = residuals.shift(1)
        formula = 'rspot ~ rfuture + lresid'
        results = smf.ols(formula, data).fit()
        print(results.summary())

                            OLS Regression Results
==============================================================================
```

```
Dep. Variable:                  rspot   R-squared:                       0.992
Model:                            OLS   Adj. R-squared:                  0.992
Method:                 Least Squares   F-statistic:                 1.473e+04
Date:                Sun, 12 Aug 2018   Prob (F-statistic):          2.65e-254
Time:                        10:58:43   Log-Likelihood:                -118.19
No. Observations:                 246   AIC:                             242.4
Df Residuals:                     243   BIC:                             252.9
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.0093      0.025      0.370      0.712      -0.040       0.059
rfuture        0.9848      0.006    170.344      0.000       0.973       0.996
lresid       -55.0602      5.785     -9.518      0.000     -66.455     -43.665
==============================================================================
Omnibus:                      119.738   Durbin-Watson:                   2.268
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              920.333
Skew:                          -1.759   Prob(JB):                    1.42e-200
Kurtosis:                      11.798   Cond. No.                     1.02e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.02e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Note that we can either include or exclude the lagged terms and either form 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 would 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.

```
In [5]: # specification 2: rspot rfutures L.rspot L.rfutures
        formula = 'rspot ~ rfuture + lspot + lfuture'
        results = smf.ols(formula, data).fit()
        print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  rspot   R-squared:                       0.992
Model:                            OLS   Adj. R-squared:                  0.992
Method:                 Least Squares   F-statistic:                     9926.
Date:                Sun, 12 Aug 2018   Prob (F-statistic):          5.83e-253
Time:                        10:58:43   Log-Likelihood:                -116.33
No. Observations:                 246   AIC:                             240.7
Df Residuals:                     242   BIC:                             254.7
```

```
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.6699      0.642      1.044      0.298      -0.594       1.934
rfuture        0.9800      0.006    171.085      0.000       0.969       0.991
lspot         55.7343      5.710      9.761      0.000      44.486      66.982
lfuture      -55.8216      5.719     -9.760      0.000     -67.087     -44.556
==============================================================================
Omnibus:                      116.793   Durbin-Watson:                   2.245
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              803.607
Skew:                           1.749   Prob(JB):                    3.15e-175
Kurtosis:                      11.135   Cond. No.                     3.33e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 3.33e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

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.

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 **'macro.pickle'** workfile. 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 the **DFGLS** function with a maximum lag length of 12.[36]

```
In [6]: from statsmodels.tsa.vector_ar import vecm

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                          'QMF Book/book Ran/data files new/Book4e_data/'
        data = pd.read_excel(abspath + 'FRED.xls', index_col=0)
```

Before specifying the VECM using the Johansen method, it is often very useful to graph the variables to see how they behave over time and with respect to each other (see Figure 31). This will also help us to select the correct option for the VECM specification, e.g., if the series appear to follow a linear trend. To generate a graph of all variables we use the well-known Python library **matplotlib.pyplot** and type the commands as in the follow code cell.

[36] Note that for the 3-year, 5-year, and 10-year rates the unit root test is rejected for the optimal lag length based on the Schwarz criterion. However, for the sake of this example we will continue to use all six of the rates.

```
In [7]: plt.figure(1)
        plt.plot(data['GS3M'], label='GS3M')
        plt.plot(data['GS6M'], label='GS6M')
        plt.plot(data['GS1'], label='GS1')
        plt.plot(data['GS3'], label='GS3')
        plt.plot(data['GS5'], label='GS5')
        plt.plot(data['GS10'], label='GS10')
        plt.legend()
        plt.xlabel('Date')
        plt.ylabel('Residuals')
        plt.grid(True)
        plt.show()
```

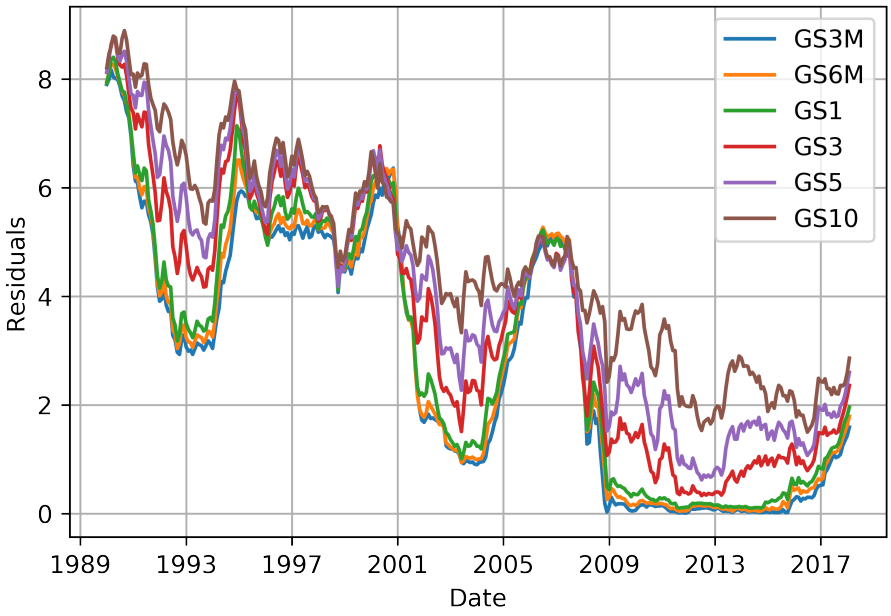


Figure 31: Graph of the Six US Treasury Interest Rates Series

We see that the series generally follow a linear downward trend, though some series show stronger inter-temporal variation with larger drops than others. Additionally, while all of the 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 to 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 methods implemented in Statsmodels' **vecm** function. To access this test, we first import the function from **statsmodels.tsa.vector_ar**. We can run the lag-order selection test. In the inputs for the test, we first define all the **six interest rates as Dependent variables** and then set up a **Maximum lag order of 12**. We then print the summary table from the model instance and the test output should appear as below.

```
In [8]: # VECM select appropriate lag orders
        model = vecm.select_order(data,maxlags=12)
        print(model.summary())
```

```
 VECM Order Selection (* highlights the minimums)
=================================================
       AIC         BIC         FPE         HQIC
-------------------------------------------------
0       -31.49     -31.00*   2.106e-14      -31.30
1      -31.80*      -30.89  1.546e-14*     -31.44*
2       -31.76      -30.44   1.604e-14      -31.23
3       -31.76      -30.01   1.612e-14      -31.06
4       -31.69      -29.52   1.739e-14      -30.82
5       -31.68      -29.10   1.751e-14      -30.65
6       -31.63      -28.63   1.843e-14      -30.44
7       -31.70      -28.27   1.738e-14      -30.33
8       -31.62      -27.77   1.895e-14      -30.08
9       -31.61      -27.35   1.917e-14      -29.91
10      -31.52      -26.84   2.114e-14      -29.66
11      -31.45      -26.35   2.295e-14      -29.42
12      -31.45      -25.94   2.321e-14      -29.25
-------------------------------------------------
```

The four information criteria provide inconclusive results regarding the optimal lag length. While the FPE, the HQIC and the AIC suggest an optimal lag length of one lag, the BIC favours a lag length of zero. Note that the difference in optimal model order could be attributed to the relatively small sample size available with this monthly sample compared with the number of observations that would have been available were daily data used, implying that the penalty term in BIC is more severe on extra parameters in this case. In the framework of this example, we follow the AIC and select a lag length of one.

The next step in fitting a VECM is determining the number of cointegrating relationships using a VEC rank test. The corresponding Python function is **select_coint_rank**. The tests for cointegration implemented in this function are based on Johansen's method 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.

To access the VEC rank test, we call the built-in function **select_coint_rank** from **vecm** and define the list of dependent variables (in this case, all six series). Next, we set the trend specification argument **det_order** being one as based upon a visual inspection of the data series, since they roughly seemed to follow a linear downward trend. Then, we also set the maximum lag to be included in the underlying VAR model to one as determined in the previous step. Finally, the λ_{trace} statistics method is selected with significant level at 1%. By executing the command **print(vec_rank1.summary())**, the following output should appear in the output window.

```
In [9]: vec_rank1 = vecm.select_coint_rank(data, det_order = 1, k_ar_diff = 1,
                                          method = 'trace', signif=0.01)
        print(vec_rank1.summary())
```

```
Johansen cointegration test using trace test statistic with 1% significance level
=====================================
r_0 r_1 test statistic critical value
-------------------------------------
  0   6          208.2          117.0
  1   6          139.3          87.77
  2   6          88.18          62.52
  3   6          46.96          41.08
  4   6          21.71          23.15
-------------------------------------
```

The first column in the table shows the rank of the VECM that has been tested or, in other words, the number of cointegrating relationships for the set of interest rates, while the second reports the number of equations in total. We find the λ_{trace} statistics in the third column, together with the corresponding critical values. The first row of the table tests the null hypothesis of at most one cointegrating vector, 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 one. The test statistic of 208.2 considerably exceeds the critical value (117.0) and so the null of at most one cointegrating vector is rejected. If we then move to the next row, the test statistic (139.3) again exceeds the critical value so that the null of at most two cointegrating vectors is also rejected. This continues, and we also reject the null of at most three cointegrating vectors, but we stop at the next row, where we do not reject the null hypothesis of at most four cointegrating vectors at the 1% level, and this is the conclusion.

Besides the λ_{trace} statistic, we can also employ an alternative statistic, the maximum-eigenvalue statistic (λ_{max}). 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. We can generate the results for this alternative test by going back to the 'select_coint_rank' inputs and changing the argument **method='trace'** to the **method='maxeig'**. We leave everything else unchanged and execute the command.

The test output should now report the results for the λ_{max} statistics in the summary table below. We find that the results differ slightly compared with our previous conclusion of five cointegrating relations between the interest rates. There are only at most four cointegrating vectors in this case.

```
In [10]: vec_rank2 = vecm.select_coint_rank(data, det_order = 1, k_ar_diff = 1,
                                            method = 'maxeig', signif=0.01)
         print(vec_rank2.summary())
```

```
Johansen cointegration test using maximum eigenvalue test statistic with
 1% significance level
=====================================
r_0 r_1 test statistic critical value
-------------------------------------
  0   1          68.92          49.41
  1   2          51.11          42.86
  2   3          41.21          36.19
  3   4          25.25          29.26
-------------------------------------
```

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 call the function **VECM**. In the VECM inputs, we first specify all six interest rates as the dependent variables and then select five as the number of cointegrating equations (rank) and one again as the maximum lag to be included in the underlying VAR model. As in the previous input, we set the **deterministic** argument to 'co', meaning that there is a constant inside the cointegrating relationship and simply run the code. The following output should appear in the output window.

```
In [11]: # VECM
         model = vecm.VECM(data, k_ar_diff=1,coint_rank=5,deterministic='co')
         res = model.fit()
         print(res.summary())
```

```
Det. terms outside the coint. relation & lagged endog. parameters for equation GS3M
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0146      0.038      0.388      0.698      -0.059       0.089
L1.GS3M        0.1942      0.148      1.311      0.190      -0.096       0.485
L1.GS6M       -0.0403      0.273     -0.148      0.883      -0.576       0.495
L1.GS1         0.0375      0.249      0.150      0.880      -0.451       0.526
L1.GS3         0.3382      0.269      1.259      0.208      -0.188       0.865
L1.GS5        -0.2019      0.326     -0.619      0.536      -0.841       0.437
L1.GS10       -0.0257      0.171     -0.150      0.880      -0.361       0.310
Det. terms outside the coint. relation & lagged endog. parameters for equation GS6M
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0407      0.038      1.069      0.285      -0.034       0.115
L1.GS3M        0.2067      0.150      1.382      0.167      -0.086       0.500
L1.GS6M       -0.0156      0.276     -0.057      0.955      -0.556       0.525
L1.GS1         0.0283      0.251      0.113      0.910      -0.465       0.521
L1.GS3         0.3511      0.271      1.295      0.195      -0.180       0.883
L1.GS5        -0.1876      0.329     -0.570      0.569      -0.833       0.458
L1.GS10       -0.0029      0.173     -0.017      0.987      -0.341       0.336
Det. terms outside the coint. relation & lagged endog. parameters for equation GS1
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0339      0.042      0.811      0.417      -0.048       0.116
L1.GS3M        0.0103      0.164      0.062      0.950      -0.312       0.332
L1.GS6M        0.2414      0.303      0.797      0.425      -0.352       0.835
L1.GS1        -0.0967      0.276     -0.350      0.726      -0.638       0.445
L1.GS3         0.4569      0.298      1.534      0.125      -0.127       1.041
L1.GS5        -0.2292      0.362     -0.634      0.526      -0.938       0.479
L1.GS10        0.0418      0.190      0.221      0.825      -0.330       0.414
Det. terms outside the coint. relation & lagged endog. parameters for equation GS3
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
```

```
------------------------------------------------------------------------------
const          0.0258      0.053      0.487      0.627      -0.078       0.130
L1.GS3M       -0.1257      0.208     -0.603      0.546      -0.534       0.283
L1.GS6M        0.0022      0.384      0.006      0.996      -0.751       0.755
L1.GS1        -0.0154      0.350     -0.044      0.965      -0.702       0.671
L1.GS3         0.7017      0.378      1.857      0.063      -0.039       1.442
L1.GS5        -0.3031      0.459     -0.661      0.509      -1.202       0.596
L1.GS10        0.0312      0.241      0.130      0.897      -0.441       0.503
Det. terms outside the coint. relation & lagged endog. parameters for equation GS5
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0436      0.054      0.806      0.420      -0.062       0.150
L1.GS3M       -0.1869      0.213     -0.879      0.379      -0.603       0.230
L1.GS6M       -0.0448      0.392     -0.114      0.909      -0.813       0.724
L1.GS1         0.0753      0.357      0.211      0.833      -0.625       0.776
L1.GS3         0.3905      0.385      1.013      0.311      -0.365       1.146
L1.GS5        -0.0344      0.468     -0.074      0.941      -0.952       0.883
L1.GS10        0.0185      0.246      0.075      0.940      -0.463       0.500
Det. terms outside the coint. relation & lagged endog. parameters for equation GS10
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0807      0.051      1.577      0.115      -0.020       0.181
L1.GS3M       -0.3120      0.201     -1.553      0.120      -0.706       0.082
L1.GS6M        0.2012      0.371      0.543      0.587      -0.525       0.928
L1.GS1         0.0042      0.338      0.012      0.990      -0.658       0.666
L1.GS3        -0.0104      0.364     -0.028      0.977      -0.724       0.704
L1.GS5         0.2449      0.442      0.554      0.580      -0.622       1.112
L1.GS10        0.0205      0.232      0.089      0.929      -0.434       0.475
                Loading coefficients (alpha) for equation GS3M
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ec1           -0.3421      0.116     -2.956      0.003      -0.569      -0.115
ec2            0.3224      0.223      1.447      0.148      -0.114       0.759
ec3           -0.1587      0.180     -0.880      0.379      -0.512       0.195
ec4            0.5228      0.183      2.856      0.004       0.164       0.882
ec5           -0.4510      0.182     -2.479      0.013      -0.807      -0.094
                Loading coefficients (alpha) for equation GS6M
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ec1           -0.0181      0.117     -0.155      0.877      -0.247       0.211
ec2           -0.1849      0.225     -0.822      0.411      -0.626       0.256
ec3            0.0706      0.182      0.388      0.698      -0.286       0.427
ec4            0.4326      0.185      2.341      0.019       0.070       0.795
ec5           -0.3774      0.184     -2.056      0.040      -0.737      -0.018
                Loading coefficients (alpha) for equation GS1
```

	coef	std err	z	P>|z|	[0.025	0.975]
ec1	0.0457	0.128	0.356	0.721	-0.206	0.297
ec2	-0.1466	0.247	-0.594	0.553	-0.631	0.337
ec3	-0.0267	0.200	-0.133	0.894	-0.419	0.365
ec4	0.3870	0.203	1.908	0.056	-0.011	0.785
ec5	-0.3263	0.202	-1.618	0.106	-0.721	0.069

Loading coefficients (alpha) for equation GS3

	coef	std err	z	P>|z|	[0.025	0.975]
ec1	0.2201	0.163	1.352	0.176	-0.099	0.539
ec2	-0.4621	0.313	-1.475	0.140	-1.076	0.152
ec3	0.2639	0.254	1.040	0.298	-0.233	0.761
ec4	0.1111	0.257	0.431	0.666	-0.394	0.616
ec5	-0.1864	0.256	-0.728	0.466	-0.688	0.315

Loading coefficients (alpha) for equation GS5

	coef	std err	z	P>|z|	[0.025	0.975]
ec1	0.3214	0.166	1.936	0.053	-0.004	0.647
ec2	-0.5487	0.320	-1.717	0.086	-1.175	0.078
ec3	0.1846	0.259	0.713	0.476	-0.323	0.692
ec4	0.3627	0.263	1.381	0.167	-0.152	0.877
ec5	-0.4491	0.261	-1.721	0.085	-0.961	0.062

Loading coefficients (alpha) for equation GS10

	coef	std err	z	P>|z|	[0.025	0.975]
ec1	0.3909	0.157	2.491	0.013	0.083	0.698
ec2	-0.6212	0.302	-2.057	0.040	-1.213	-0.029
ec3	0.1687	0.245	0.690	0.490	-0.311	0.648
ec4	0.3948	0.248	1.590	0.112	-0.092	0.881
ec5	-0.4355	0.247	-1.765	0.077	-0.919	0.048

Cointegration relations for loading-coefficients-column 1

	coef	std err	z	P>|z|	[0.025	0.975]
beta.1	1.0000	0	0	0.000	1.000	1.000
beta.2	8.918e-16	0	0	0.000	8.92e-16	8.92e-16
beta.3	-2.547e-16	0	0	0.000	-2.55e-16	-2.55e-16
beta.4	-2.707e-17	0	0	0.000	-2.71e-17	-2.71e-17
beta.5	4.067e-16	0	0	0.000	4.07e-16	4.07e-16
beta.6	-1.2273	0.211	-5.828	0.000	-1.640	-0.815

Cointegration relations for loading-coefficients-column 2

	coef	std err	z	P>|z|	[0.025	0.975]

```
beta.1      4.501e-16          0          0      0.000     4.5e-16     4.5e-16
beta.2         1.0000          0          0      0.000       1.000       1.000
beta.3     -6.735e-16          0          0      0.000   -6.74e-16   -6.74e-16
beta.4     -1.835e-16          0          0      0.000   -1.83e-16   -1.83e-16
beta.5     -5.765e-16          0          0      0.000   -5.77e-16   -5.77e-16
beta.6        -1.2516      0.211     -5.922      0.000      -1.666      -0.837
          Cointegration relations for loading-coefficients-column 3
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
beta.1      5.866e-16          0          0      0.000    5.87e-16    5.87e-16
beta.2     -1.321e-16          0          0      0.000   -1.32e-16   -1.32e-16
beta.3         1.0000          0          0      0.000       1.000       1.000
beta.4      2.364e-16          0          0      0.000    2.36e-16    2.36e-16
beta.5      8.786e-16          0          0      0.000    8.79e-16    8.79e-16
beta.6        -1.2643      0.196     -6.452      0.000      -1.648      -0.880
          Cointegration relations for loading-coefficients-column 4
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
beta.1     -4.297e-17          0          0      0.000    -4.3e-17    -4.3e-17
beta.2      4.312e-16          0          0      0.000    4.31e-16    4.31e-16
beta.3     -2.667e-16          0          0      0.000   -2.67e-16   -2.67e-16
beta.4         1.0000          0          0      0.000       1.000       1.000
beta.5     -1.761e-16          0          0      0.000   -1.76e-16   -1.76e-16
beta.6        -1.2579      0.129     -9.769      0.000      -1.510      -1.005
          Cointegration relations for loading-coefficients-column 5
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
beta.1      1.204e-18          0          0      0.000     1.2e-18     1.2e-18
beta.2      2.388e-16          0          0      0.000    2.39e-16    2.39e-16
beta.3     -1.487e-16          0          0      0.000   -1.49e-16   -1.49e-16
beta.4     -2.183e-17          0          0      0.000   -2.18e-17   -2.18e-17
beta.5         1.0000          0          0      0.000       1.000       1.000
beta.6        -1.1693      0.073    -15.966      0.000      -1.313      -1.026
==============================================================================
```

Jupyter produces a large set of tables, with each panel presenting its header above. The header contains information about the test: the parameters of each equation, and the deterministic terms regarding the overall model. The first six panels contain estimates of the short-run parameters along with their standard errors, z-statistics, and confidence intervals.

Additionally, there are loading coefficient panels for each of the five cointegrating vectors for this model, including both α and β. The five coefficients on 'ec1', 'ec2', 'ec3', 'ec4', 'ec5' are the parameters in the adjustment matrix $\alpha(\beta)$, and the next several columns contain the estimated parameters of the cointegrating vector for this model, along with their standard errors, z-statistics, and confidence intervals.

19 Volatility modelling

19.1 Testing for 'ARCH effects' in exchange rate returns

Reading: Brooks (2019, section 9.7)
In this section we will test for 'ARCH effects' in exchange rates using the '**currrencies.pickle**' dataset. First, we want to compute the (Engle, 1982) test for ARCH effects to make sure that this class of models is appropriate for the data. This exercise (and the remaining 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.

```
In [1]: import pickle
        from statsmodels.stats.diagnostic import het_arch
        from statsmodels.compat import lzip
        import statsmodels.api as sm

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                          'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'currencies.pickle', 'rb') as handle:
            data = pickle.load(handle)
```

A test for the presence of ARCH in the residuals is calculated by regressing the squared residuals on a constant and p lags, where p is set by the user. As an example, assume that p is set to five. The first step is to estimate a linear model so that the residuals can be tested for ARCH. In Jupyter, we perform these tests by fitting a constant-only model based on an OLS regression and testing for ARCH effects using Engle's Lagrange multiplier test. To do so, we exercise the command in the following code cell. Specifically, we generate a set of new variables (the lagged values of 'rgbp' from one period to five periods). An OLS regression instance is then constructed by regressing 'rgbp' on its own lagged value plus a constant term. Finally, we obtain the residual series by typing the command **results.resid**.

```
In [2]: data1 = sm.add_constant(data['rgbp'])
        results = sm.OLS(data1['rgbp'],data1['const']).fit()
        print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   rgbp   R-squared:                       0.000
Model:                            OLS   Adj. R-squared:                  0.000
Method:                 Least Squares   F-statistic:                       nan
Date:                Wed, 22 Aug 2018   Prob (F-statistic):                nan
Time:                        10:56:04   Log-Likelihood:                -3952.8
No. Observations:                7141   AIC:                             7908.
Df Residuals:                    7140   BIC:                             7914.
Df Model:                           0
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
```

```
const          0.0035      0.005      0.695      0.487      -0.006       0.013
==============================================================================
Omnibus:                     2045.985   Durbin-Watson:                   1.620
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            50171.502
Skew:                           0.813   Prob(JB):                         0.00
Kurtosis:                      15.883   Cond. No.                         1.00
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

To test for ARCH effects in the residuals, we import the function **het_arch** from **statsmodels.stats.diagnostic**. In the input brackets, we only need to feed the residual series and specify the list of lag orders as five. As can be seen from the test output, the Engle test is based on the null hypothesis that there are no ARCH effects against the alternative hypothesis that the data is characterised by (in our case) ARCH(5) disturbances.

```
In [3]: res = het_arch(results.resid,maxlag=5)
        name = ['lm','lmpval','fval','fpval']
        lzip(name,res)

Out[3]: [('lm', 252.99550035192351),
         ('lmpval', 1.2512748060596812e-52),
         ('fval', 52.414840571481349),
         ('fpval', 1.4008125355474366e-53)]
```

The test shows a p-value of 0.0000, which is well below 0.05, suggesting the presence of ARCH effects in the pound-dollar returns.

19.2 Estimating GARCH models

Reading: Brooks (2019, section 9.9)

To estimate a GARCH-type model in Python, we select the **arch_model** function from **arch**. In the function inputs, we define **Dependent variable: rjpy**. We do not include any further independent variables but instead continue by specifying the main model specification. Let us first determine the type of volatility model. In this case, we set the argument as **vol='GARCH'**. This means that the model would include one α and one β term (i.e., one lag of the squared errors and one lag of the conditional variance, respectively). Meanwhile, it is good to clarify that the maximum lags with respect to the ARCH and GARCH terms is one by default, which corresponds to the p and q parameters.[37]

```
In [1]: import pickle
        from arch import arch_model

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'currencies.pickle', 'rb') as handle:
            data = pickle.load(handle)
```

```
In [2]: # The default set of options produces a model with a constant mean,
        # GARCH(1,1) conditional variance and normal errors.
        am = arch_model(data['rjpy'], vol='GARCH')
        res = am.fit()
        print(res.summary())
```

```
Iteration:      1,   Func. Count:      6,   Neg. LLF: 4358.469077544289
Iteration:      2,   Func. Count:     17,   Neg. LLF: 4357.512862516335
Iteration:      3,   Func. Count:     24,   Neg. LLF: 4354.350593693825
Iteration:      4,   Func. Count:     31,   Neg. LLF: 4349.030055343708
Iteration:      5,   Func. Count:     38,   Neg. LLF: 4344.583822901452
Iteration:      6,   Func. Count:     45,   Neg. LLF: 4343.147305239814
Iteration:      7,   Func. Count:     53,   Neg. LLF: 4342.831776832556
Iteration:      8,   Func. Count:     61,   Neg. LLF: 4342.692628665991
Iteration:      9,   Func. Count:     68,   Neg. LLF: 4341.537067194463
Iteration:     10,   Func. Count:     76,   Neg. LLF: 4341.453260751554
Iteration:     11,   Func. Count:     83,   Neg. LLF: 4340.953920529477
Iteration:     12,   Func. Count:     90,   Neg. LLF: 4340.9227508137765
Iteration:     13,   Func. Count:     97,   Neg. LLF: 4340.773793894918
Iteration:     14,   Func. Count:    103,   Neg. LLF: 4340.771663888115
Iteration:     15,   Func. Count:    109,   Neg. LLF: 4340.7715569509155
Iteration:     16,   Func. Count:    115,   Neg. LLF: 4340.771554320768
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 4340.771554314696
            Iterations: 16
            Function evaluations: 115
            Gradient evaluations: 16
                      Constant Mean - GARCH Model Results
```

[37] Apart from the p and q parameters for GARCH modelling, the parameter o sets the lag order of the asymmetric innovation. This term is zero by default; however, will be altered for modelling EGARCH in the later section.

```
==========================================================================
Dep. Variable:                    rjpy   R-squared:                -0.000
Mean Model:             Constant Mean   Adj. R-squared:           -0.000
Vol Model:                      GARCH   Log-Likelihood:          -4340.77
Distribution:                  Normal   AIC:                      8689.54
Method:            Maximum Likelihood   BIC:                      8717.04
                                        No. Observations:            7141
Date:                Wed, Aug 22 2018   Df Residuals:                7137
Time:                        14:06:15   Df Model:                       4
                            Mean Model
==========================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
--------------------------------------------------------------------------
mu         6.2772e-03  5.274e-03      1.190      0.234 [-4.059e-03,1.661e-02]
                          Volatility Model
==========================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
--------------------------------------------------------------------------
omega      1.5695e-03  7.055e-04      2.225  2.611e-02 [1.867e-04,2.952e-03]
alpha[1]       0.0353  8.973e-03      3.930  8.487e-05 [1.768e-02,5.285e-02]
beta[1]        0.9581  9.875e-03     97.025      0.000     [  0.939,  0.977]
==========================================================================

Covariance estimator: robust
```

The **arch_model** provides various options regarding how to vary the model. You can examine the parameters by looking through the documentation. The **vol** argument can be used to choose the type of volatility modelling (see later in this section), while the argument **dist** provides different options for the assumed distribution of the errors, e.g., instead of applying a Normal distribution (default setting) we can specify a Student's t-distribution.

Estimating the GARCH(1,1) model for the yen-dollar ('rjpy') series using the instructions as listed above and the default settings elsewhere would yield the table of results above.

The coefficients on both the lagged squared residuals and lagged conditional variance terms in the conditional variance equation (i.e., the third panel in the output subtitled 'ARCH') are highly statistically significant. 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. The individual conditional variance coefficients are also as one would expect. The variance intercept term omega in the 'ARCH' panel is very small, and the 'ARCH'-parameter 'alpha[1]' is around 0.04 while the coefficient on the lagged conditional variance 'beta[1]' is larger, at 0.96.

19.3 GJR and EGARCH 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 in Python, the GJR and EGARCH 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.

We start by estimating the EGARCH model. To do so, we need to change the argument **vol='GARCH'** to **vol='EGARCH'** for the **arch_model** inputs, while keeping other parameters unchanged. Moreover, we are now asked to provide the maximum number of lags for the asymmetric innovation because the function initialises with only one **arch** and one **grach** term. To start with, we choose **o=1** to resemble the previous classic GARCH model.

```
In [3]: # E-GARCH
        am = arch_model(data['rjpy'], vol='EGARCH',o=1)
        res = am.fit()
        print(res.summary())
```

```
Iteration:      1,   Func. Count:      7,   Neg. LLF: 4397.486452434722
Iteration:      2,   Func. Count:     23,   Neg. LLF: 4362.398722296345
Iteration:      3,   Func. Count:     35,   Neg. LLF: 4355.522829279152
Iteration:      4,   Func. Count:     46,   Neg. LLF: 4348.432196558825
Iteration:      5,   Func. Count:     57,   Neg. LLF: 4331.514196657024
Iteration:      6,   Func. Count:     65,   Neg. LLF: 4326.906644217339
Iteration:      7,   Func. Count:     73,   Neg. LLF: 4324.506317267667
Iteration:      8,   Func. Count:     84,   Neg. LLF: 4321.488049820839
Iteration:      9,   Func. Count:     91,   Neg. LLF: 4320.941701170441
Iteration:     10,   Func. Count:     98,   Neg. LLF: 4320.6562194010285
Iteration:     11,   Func. Count:    105,   Neg. LLF: 4320.641194704814
Iteration:     12,   Func. Count:    112,   Neg. LLF: 4320.639816946357
Iteration:     13,   Func. Count:    119,   Neg. LLF: 4320.6397432610665
Iteration:     14,   Func. Count:    126,   Neg. LLF: 4320.639640946247
Iteration:     15,   Func. Count:    134,   Neg. LLF: 4320.639625868453
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 4320.639625250302
            Iterations: 15
            Function evaluations: 135
            Gradient evaluations: 15
                    Constant Mean - EGARCH Model Results
==============================================================================
Dep. Variable:                    rjpy   R-squared:                      -0.000
Mean Model:             Constant Mean   Adj. R-squared:                 -0.000
Vol Model:                      EGARCH   Log-Likelihood:               -4320.64
Distribution:                   Normal   AIC:                           8651.28
Method:            Maximum Likelihood   BIC:                           8685.65
                                        No. Observations:                 7141
Date:                Wed, Aug 22 2018   Df Residuals:                     7136
Time:                        14:06:15   Df Model:                            5
                                  Mean Model
```

```
================================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
--------------------------------------------------------------------------------
mu         4.2062e-03  1.159e-03      3.629  2.844e-04 [1.935e-03,6.478e-03]
                              Volatility Model
================================================================================
                 coef    std err          t      P>|t|        95.0% Conf. Int.
--------------------------------------------------------------------------------
omega         -0.0107  7.226e-03     -1.486      0.137  [-2.490e-02,3.424e-03]
alpha[1]       0.1006  1.470e-02      6.841  7.849e-12     [7.177e-02,  0.129]
gamma[1]      -0.0272  1.053e-02     -2.583  9.794e-03 [-4.784e-02,-6.560e-03]
beta[1]        0.9868  4.794e-03    205.840      0.000       [  0.977,  0.996]
================================================================================

Covariance estimator: robust
```

Looking at the results, we find that all EARCH and EGARCH terms are statistically significant except the constant. The EARCH terms represent the influence of news – lagged innovations – in the Nelson (1991) EGARCH model. The term 'gamma[1]' captures the following:

$$\frac{v_{t-1}}{\sqrt{\sigma_{t-1}^2}} \tag{12}$$

and 'alpha[1]' captures:

$$\frac{|v_{t-1}|}{\sqrt{\sigma_{t-1}^2}} - \sqrt{\frac{2}{\pi}} \tag{13}$$

The former is a typical ARCH effect(i.e., the sign effect) while the latter determines an asymmetric effect (in other words, the size effect). It is evident that the positive estimate on the asymmetric effect implies that negative shocks result in a higher next-period conditional variance than positive shocks of the same sign. The result for the EGARCH asymmetry term is consistent with what would have been expected in the case of the application of a GARCH model to a set of stock returns. Indeed, both the *leverage effect* and *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 lower next-period volatility than when the yen strengthens by the same amount.

```
In [4]: # GJR-GARCH
        am = arch_model(data['rjpy'], p=1, o=1, q=1, vol='GARCH')
        res = am.fit()
        print(res.summary())
```

```
Iteration:      1,   Func. Count:      7,   Neg. LLF: 4351.036676266833
Iteration:      2,   Func. Count:     19,   Neg. LLF: 4350.145546474399
Iteration:      3,   Func. Count:     28,   Neg. LLF: 4347.873921690083
Iteration:      4,   Func. Count:     38,   Neg. LLF: 4347.520571366731
Iteration:      5,   Func. Count:     46,   Neg. LLF: 4341.0326487152015
```

```
Iteration:      6,   Func. Count:     54,   Neg. LLF: 4337.01501415516
Iteration:      7,   Func. Count:     62,   Neg. LLF: 4333.740068844954
Iteration:      8,   Func. Count:     70,   Neg. LLF: 4331.051027886871
Iteration:      9,   Func. Count:     78,   Neg. LLF: 4328.811764930388
Iteration:     10,   Func. Count:     86,   Neg. LLF: 4328.348020964243
Iteration:     11,   Func. Count:     94,   Neg. LLF: 4327.258888584185
Iteration:     12,   Func. Count:    103,   Neg. LLF: 4327.169530282161
Iteration:     13,   Func. Count:    111,   Neg. LLF: 4326.593301728237
Iteration:     14,   Func. Count:    118,   Neg. LLF: 4326.5409951654965
Iteration:     15,   Func. Count:    125,   Neg. LLF: 4326.536292415701
Iteration:     16,   Func. Count:    132,   Neg. LLF: 4326.535882616242
Iteration:     17,   Func. Count:    139,   Neg. LLF: 4326.535853904752
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 4326.535853905975
            Iterations: 17
            Function evaluations: 139
            Gradient evaluations: 17
                  Constant Mean - GJR-GARCH Model Results
==============================================================================
Dep. Variable:                    rjpy   R-squared:                      -0.000
Mean Model:             Constant Mean   Adj. R-squared:                 -0.000
Vol Model:                  GJR-GARCH   Log-Likelihood:               -4326.54
Distribution:                  Normal   AIC:                            8663.07
Method:            Maximum Likelihood   BIC:                            8697.44
                                        No. Observations:                  7141
Date:                Wed, Aug 22 2018   Df Residuals:                      7136
Time:                        14:06:15   Df Model:                             5
                            Mean Model
==============================================================================
                 coef    std err          t      P>|t|       95.0% Conf. Int.
------------------------------------------------------------------------------
mu         3.5586e-03  5.105e-03      0.697      0.486 [-6.448e-03,1.357e-02]
                          Volatility Model
==============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
------------------------------------------------------------------------------
omega      1.9666e-03  8.585e-04      2.291  2.197e-02 [2.840e-04,3.649e-03]
alpha[1]       0.0268  6.298e-03      4.259  2.050e-05 [1.448e-02,3.917e-02]
gamma[1]       0.0254  1.104e-02      2.302  2.133e-02 [3.777e-03,4.705e-02]
beta[1]        0.9520  9.685e-03     98.306      0.000     [  0.933,  0.971]
==============================================================================

Covariance estimator: robust
```

Let us now test a GJR model. For this, we come back to the original set-up: the volatility model **vol='GARCH'**. In the GJR inputs, we specify **1 ARCH maximum lag**, **1 TARCH maximum lag** and **1 GARCH maximum lag**, and press **SHIFT** and **ENTER** to fit the model.[38]

[38] Note that the p and q parameters can be ignored because the function by default sets them both to be one.

Similar to the EGARCH model, we find that all of the ARCH, TARCH and GARCH terms are statistically significant. The 'gamma[1]' term captures the $v_{t-1}^2 I_{t-1}$ term where $I_{t-1} = 1$ if $v_{t-1} < 0$ and $I_{t-1} = 0$ otherwise. We find a positive coefficient estimate on the 'gamma[1]' term, which is again consistent with what we would expect to find according to the *leverage effect* explanation if we were modelling stock return volatilities.

Apart from these two extensions, the GARCH-in-mean model has been widely implemented in the situation where investors should be rewarded for taking additional risk. One way to incorporate this idea is to add the conditional variance term back into the conditional mean equation. Unfortunately, there is no built-in function in Python to estimate such a model; this implementation is therefore left for the time being.

19.4 Forecasting from GARCH models

Reading: Brooks (2019, section 9.18)

GARCH-type models can be used to forecast volatility. In this sub-section, we will focus on generating conditional variance forecasts using Python. Let us assume that we want to generate forecasts based on the GARCH model estimated earlier for the forecast period 03 Aug 2016 to 03 Jul 2018. The first step is to re-estimate the GARCH model for the subsample running until 02 Aug 2016. To estimate the model, we input the same specifications as previously, i.e., the 'rjpy' series from the 'currencies.pickle' workfile. However, now we only want to estimate the model for a subperiod of the data so we slice the DataFrame and generate two new series **data_in_the_sample** and **data_out_of_the_sample**, respectively, using the built-in function **loc** with break date '2016-08-02'.

```
In [1]: import pickle
        from arch import arch_model
        import matplotlib.pyplot as plt
        import NumPy as np
        import Pandas as pd

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'currencies.pickle', 'rb') as handle:
            data = pickle.load(handle)

        # Sampling
        data_in_the_sample = data.loc[:'2016-08-02','rjpy']
        data_out_of_the_sample = data.loc['2016-08-03':,'rjpy']
```

There is a variety of predicted values that we can produce, including fixed windows, rolling windows, recursive windows, static and dynamic forecasting. As was the case for the previous forecast exercise, we can either create static (a series of rolling single-step-ahead) forecasts or dynamic (multiple-step-ahead) forecasts. Unfortunately, there is no built-in function to implement those two types of forecasts. However, we can try to generate these series with our own code after the intuition behind them has been understood.

Let us start with the static forecasts. To begin with, a GARCH model instance **am** is constructed. Given that static forecasting is the type where the prediction always uses previous actual observations, a **for** loop is created to construct a rolling one-step horizon for out-of-the-sample forecasts. Specifically, we take each index datetime individually from **data_out_of_the_sample** and generate a GARCH result instance **res** based on the subsample until that date. For example, if the index datetime is '2016-08-03', the sample data for GARCH result starts from '1998-12-15' to '2016-08-03'. The one-step ahead forecasts subsequently set out after the ending date (i.e., '2016-08-03'). Likewise, the forecasts will start from '2016-08-04' if the index datetime is '2016-08-04', and so on. Note that we input '2016-08-03' as the first date in the argument **last_obs = date** because the **arch_model** does not include the last observation in the volatility modelling process.

To access the one-step forecasting value of each GARCH specification, we type the command **forecasts = res.forecast(horizon=1)**. Finally, we keep the value by the command **cvar_rjpy_stat[date] = forecasts_res.iloc[1]**. Once the loop completes, the static forecasts can be seen in the dictionary **cvar_rjpy_stat**, which later converts to a DataFrame for better presentation.

```
In [2]: am = arch_model(data['rjpy'], vol='Garch')
```

```
        cvar_rjpy_stat = {}
        for date in data_out_of_the_sample.index:
            res = am.fit(last_obs = date, disp='off')
            forecasts = res.forecast(horizon=1)
            forecasts_res = forecasts.variance.dropna()
            cvar_rjpy_stat[date] = forecasts_res.iloc[1]

        cvar_rjpy_stat = pd.DataFrame(cvar_rjpy_stat).T
```

On the other hand, it is relatively easy to obtain dynamic forecasts. By definition, we generate a GARCH result that ends on '2016-08-02' and perform a multiple-step prediction. This can be achieved by the command **forecasts = res.forecast(horizon=len(data_out_of_the_sample))**. We then create a new variable **cvar_rjpy_dyn** to save the series.

```
In [3]: res = am.fit(last_obs = '2016-08-03', disp='off')
        forecasts = res.forecast(horizon=len(data_out_of_the_sample))
        forecasts_res = forecasts.variance.dropna()

        cvar_rjpy_dyn = pd.DataFrame(data = forecasts_res.iloc[1].values,\
                                     columns=['dynamic forecasting'],\
                                     index=data_out_of_the_sample.index)
```

Finally we want to graphically examine the conditional variance forecasts. To generate a time-series graph of the static and dynamic forecasts, we create a figure object 1 and plot two series with their own labels. We can also set up x, y label and legend options. After typing the **plt.show()**, the following graph (Figure 32) should appear after executing the code cell [4].

```
In [4]: plt.figure(1)
        plt.plot(cvar_rjpy_stat, label='static forecast')
        plt.plot(cvar_rjpy_dyn, label='dynamic forecast')

        plt.xlabel('Date')
        plt.ylabel('Forecasts')
        plt.legend()
        plt.show()
```

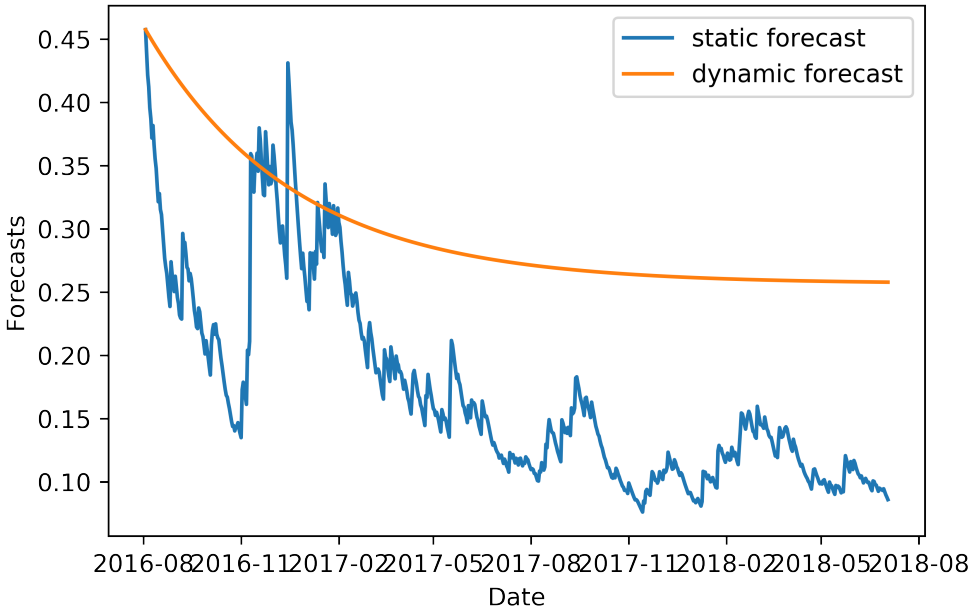


Figure 32: Dynamic and Static Forecasts

What do we observe? For the dynamic forecasts (red line), the value of the conditional variance starts from a historically low level at the end of the estimation period, relative to its unconditional average. Therefore the forecasts converge upon their long-term mean value from below as the forecast horizon increases. Turning to the static forecasts (blue line), it is evident that the variance forecasts have one large spike in early 2017. After a period of relatively high conditional variances in the first half of 2017, the values stabilise and enter a phase of historically quite low variance in the second half of 2017. 2018 sees a large rise in the conditional variances and they remain at a relatively high level for the rest of the sample period. Since in the case of the static forecasts we are looking at a series of rolling one-step ahead forecasts for the conditional variance, the values show much more volatility than those for the dynamic forecasts.

Predictions can be similarly produced for any member of the GARCH family that is estimable with the module. For specifics of how to generate predictions after specific GARCH or ARCH models, please refer to the corresponding documentation in the ARCH package.

20 Modelling seasonality in financial data

20.1 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.pickle**' 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 in Python to first convert the index of **data** DataFrame from string literals to the datetime type. The benefits of the datetime data type are various: it contains a number of built-in functions which can easily fit the purpose, such as selecting dates, calculating dates, etc. To do so, we type the following command into the code cell [2]. **data.index = pd.to_datetime(data.index, format='%Y%m%d')** where **pd.to_datetime** is the Pandas built-in function for converting the data type, while the inputs tell Python to work on the DataFrame index 'data.index' and convert dates based on the ordering of year, month and date.

Next, we generate a new variable **data['JANDUM']** by applying the function **np.where**. This new series will return the value of one conditional upon its date being in January and zero otherwise. To set up this condition, the datetime data type built-in method **month** can extract the month component from the index and return the corresponding numeric. For example, the returned numeric would be 1 if its index datetime is in January. Likewise, we work on the other two variables by the same function. It is worth noting that the input datetime parameters '2000-04-01' and '2000-12-01' need to be set as datetime types in order to match the datetime type index.

```
In [1]: import Pandas as pd
        import NumPy as np
        import statsmodels.formula.api as smf
        import pickle
        from datetime import datetime

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        with open(abspath + 'macro.pickle', 'rb') as handle:
            data = pickle.load(handle)

        data = data.dropna() # drop the missing values for some columns

In [2]: # create January dummy variable
        data.index = pd.to_datetime(data.index, format='%Y%m%d')

        data['JANDUM'] = np.where(data.index.month == 1, 1, 0)

        data['APR00DUM'] = np.where(data.index == datetime(2000,4,1), 1, 0)
        data['DEC00DUM'] = np.where(data.index == datetime(2000,12,1), 1, 0)
```

We can now run the APT-style regression first used in section 7 but this time including the new 'JANDUM' dummy variable. The formula statement for this regressions is as follows:

```
In [3]: # regression
        formula = 'ermsoft ~ ersandp + dprod + dcredit + dinflation + dmoney \
                             + dspread + rterm + APR00DUM + DEC00DUM + JANDUM'
```

```
results = smf.ols(formula, data).fit()
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 ermsoft   R-squared:                       0.412
Model:                             OLS   Adj. R-squared:                  0.396
Method:                  Least Squares   F-statistic:                     26.04
Date:                 Mon, 13 Aug 2018   Prob (F-statistic):           2.15e-37
Time:                         21:55:24   Log-Likelihood:                -1307.8
No. Observations:                 383   AIC:                             2638.
Df Residuals:                     372   BIC:                             2681.
Df Model:                          10
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      1.0442      0.494      2.113      0.035       0.073       2.016
ersandp        1.2461      0.090     13.845      0.000       1.069       1.423
dprod         -0.2849      0.703     -0.405      0.685      -1.667       1.097
dcredit       -0.0103      0.026     -0.395      0.693      -0.062       0.041
dinflation     0.2301      1.369      0.168      0.867      -2.461       2.921
dmoney         0.0030      0.016      0.190      0.849      -0.028       0.033
dspread        1.1880      3.958      0.300      0.764      -6.595       8.971
rterm          4.2093      1.635      2.574      0.010       0.994       7.425
APR00DUM     -37.7976      7.561     -4.999      0.000     -52.664     -22.931
DEC00DUM     -28.8623      7.521     -3.838      0.000     -43.650     -14.074
JANDUM         3.1391      1.654      1.898      0.059      -0.114       6.392
==============================================================================
Omnibus:                       17.848   Durbin-Watson:                   2.067
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               24.510
Skew:                           0.381   Prob(JB):                     4.76e-06
Kurtosis:                       3.978   Cond. No.                         561.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

As can be seen, the dummy is just outside being statistically significant at the 5% level, and it has the expected positive sign. The coefficient value of 3.1391 suggests that, on average and holding everything else equal, Microsoft stock returns are around 3% higher in January than the average for other months of the year.

20.2 Estimating Markov switching models

Reading: Brooks (2019, sections 10.5–10.7)

In this subsection, we will be estimating a Markov switching model in Python. The example that we will consider in this subsection relates to the changes in house prices series used previously. So we re-open **ukhp.xls**. The Statsmodel package enables us to fit two types of Markov switching models: Markov switching dynamic regression (MSDR) models, which allow a quick adjustment after the process changes state, and Markov switching autoregression (MSAR) models, that allow a more gradual adjustment. In this example, we will focus on the former case.

To call the function for Markov switching regressions, we first import the module **statsmodels.api**. In the inputs of the **MarkovRegression**, we select the dependent variable **dhp**. We want to estimate a simple switching model with just a varying intercept in each state. As the function automatically includes the (state-dependent) intercept, we do not need to specify any further variables. However, if we wanted to include additional variables that allow for either changing or non-changing coefficient parameters across states, we could do this using the respective parameters. Let us move on to specifying the number of states. The argument would be **k_regimes=2**. Finally, we also want the variance parameters to vary across states, so we type the argument **switching_variance=True**. Once all of these parameter are set, the model will resemble in the first line of the code cell [2].

We perform the regression as usual and the results appear as in the following window. Examining the findings, it is clear that the model has successfully captured the features of the data. Two distinct regimes have been identified: regime 1 has a high average price increase of 0.78% per month and a much lower standard deviation, whereas regime 2 has a negative mean return (corresponding to a price fall of 0.24% per month) and a relatively high volatility.

```
In [1]: import Pandas as pd
        import statsmodels.api as sm
        import matplotlib.pyplot as plt
        import NumPy as np

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        data = pd.read_excel(abspath + 'UKHP.xls',index_col=[0])

        data['dhp'] = data['Average House Price'].\
                           transform(lambda x : (x - x.shift(1))/x.shift(1)*100)

        data = data.dropna()

In [2]: model = sm.tsa.MarkovRegression(data['dhp'], k_regimes=2,
                                         switching_variance=True)
        res = model.fit()
        print(res.summary())
```

```
                        Markov Switching Model Results
==============================================================================
Dep. Variable:                    dhp   No. Observations:                  326
Model:              MarkovRegression   Log Likelihood                -473.592
Date:               Tue, 14 Aug 2018   AIC                            959.185
Time:                       10:45:12   BIC                            981.906
```

```
Sample:                    02-01-1991   HQIC                          968.252
                         - 03-01-2018
Covariance Type:               approx
                           Regime 0 parameters
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.7810      0.073     10.690      0.000       0.638       0.924
sigma2         0.8031      0.080     10.051      0.000       0.647       0.960
                           Regime 1 parameters
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.2388      0.137     -1.739      0.082      -0.508       0.030
sigma2         1.4167      0.220      6.436      0.000       0.985       1.848
                        Regime transition parameters
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
p[0->0]        0.9831      0.011     87.895      0.000       0.961       1.005
p[1->0]        0.0306      0.021      1.447      0.148      -0.011       0.072
==============================================================================

Warnings:
[1] Covariance matrix calculated using numerical (complex-step) differentiation.
```

We can also estimate the duration of staying in each regime. To do so, we simply call the built-in method **expected_durations** from the regression instance. We should find the following output displayed in the window below.

```
In [3]: print(res.expected_durations)

[ 59.29472979  32.66244744]
```

We find that the average duration of staying in regime 1 is 59 months and of staying in regime 2 is 33 months.

Finally, we would like to predict the probabilities of being in each particular regime. 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. To generate the state probabilities, we directly call the built-in method **smoothed_marginal_probabilities**. To visually inspect the probabilities, we can make a graph of them by creating a figure object. In the input of the function **plot**, we set the smoothed probabilities of being in the high regime as the parameter. Then we execute this chunk of code and the graph as shown in the window below should appear (see Figure 33). Examining how the graph moves over time, the probability of being in regime 1 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 1) 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 1 until early 2013 when the market again appeared to have turned a corner.

```
In [4]: plt.figure(1)
        plt.plot(res.smoothed_marginal_probabilities[1])
        plt.title('Smoothed State probabilities')
        plt.show()
```

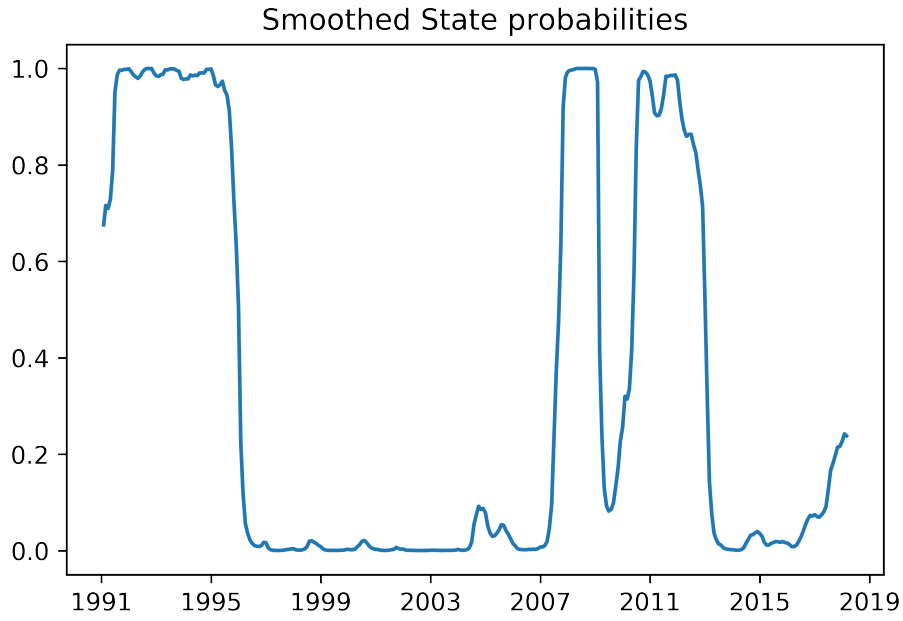


Figure 33: Smoothed State Probabilities

21 Panel data models

Reading: Brooks (2019, chapter 11)

Estimation of panel models with either fixed or random effects is very easy to implement in Python. To support these applications, there are two major libraries that can recognise that you have a panel of data and can apply the techniques accordingly: i.e., **Pandas** and **linearmodels**.

The application to be considered here is that of a variant on an early test of the capital asset pricing model (CAPM) 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 right-hand side, 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 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.[39]

To test this model, we will use the '**panelx.xls**' workfile. Let us first have a look at the data stored in a DataFrame. Call the Pandas built-in function **describe()** to create summary statistics of the data. If we apply this function for the two variables 'beta' and 'ret', the summary statistics would be generated as in the following output.

We see that missing values for the 'beta' and 'return' series are indicated by a '**nan**' in the DataFrame. Since the regression cannot be performed on the data with missing observations, we will need to "drop nan" for the dataset first in order for Python to correctly perform the regression. To do so, we apply the Pandas function **dropna** as usual.

It is necessary to set up an index column of the DataFrame **data** especially for panel regression analysis. One might examine the code demonstrated below and find that the argument **index_col=[0]** is missing in the Pandas **read_excel** function. We implement the command in this way in order to set up a multi-index for the panel regression subsequently. Note that it is important to set up both the entity and time as indices because the application of fixed-effects and/or random-effects needs these coordinates. To do so, we type the command **data = data.set_index(['firm_ident','year'])** with two column names specified.

Now our dataset is ready to be used for panel data analysis. You will need to install the third-party package **linearmodels** that has many tools specifically for panel data. For example, you can import the regression function **PooledOLS**, **PanelOLS**, **RandomEffects** and so on.

```
In [1]: import Pandas as pd
        from linearmodels import PooledOLS, PanelOLS, RandomEffects

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
```

[39] 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 Python. No doubt readers, with access to a wider range of data, will be able to think of much better applications.

```
                        'QMF Book/book Ran/data files new/Book4e_data/'
data = pd.read_excel(abspath + 'panelx.xls')

# Note: can not use 'return' as a variable name as it is one of
# Python keywords
data=data.rename(columns={'return':'ret'})
```

```
In [2]: # data summary
        data['ret'].describe()
```

```
Out[2]: count    24091.000000
        mean        -0.001546
        std          0.038328
        min         -1.005126
        25%         -0.004813
        50%         -0.003933
        75%         -0.002967
        max          0.706354
        Name: ret, dtype: float64
```

```
In [3]: data['beta'].describe()
```

```
Out[3]: count    9073.000000
        mean        1.104948
        std         0.203569
        min         0.660871
        25%         0.940652
        50%         1.081632
        75%         1.248908
        max         1.611615
        Name: beta, dtype: float64
```

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 **PooledOLS** function for simple OLS models. In particular, we type the following regression formula 'ret $\sim$ 1 + beta' into the built-in **from_formula** inputs.[40]

```
In [4]: # dropna, Note: regression can not be performed if nan exists in the dataset
        data = data.dropna()

        # set up a multi-index
        data = data.set_index(['firm_ident','year'])

        # Simple Pooled Regression
        model = PooledOLS.from_formula('ret ~ 1 + beta',data)
        res = model.fit()
        print(res)
```

[40] The design of **linearmodels** differs from **statsmodels** as the latter will automatically include a constant term into the formula. Therefore, we need to manually enter 1 if we want to add an intercept into the regression specification in **linearmodels**.

```
                          PooledOLS Estimation Summary
================================================================================
Dep. Variable:                    ret   R-squared:                      3.118e-06
Estimator:                  PooledOLS   R-squared (Between):              -0.0087
No. Observations:                8856   R-squared (Within):            -9.129e-05
Date:                Tue, Aug 14 2018   R-squared (Overall):            3.118e-06
Time:                        14:12:14   Log-likelihood                  1.357e+04
Cov. Estimator:            Unadjusted
                                        F-statistic:                      0.0276
Entities:                        1734   P-value                           0.8680
Avg Obs:                       5.1073   Distribution:                  F(1,8854)
Min Obs:                       1.0000
Max Obs:                       11.000   F-statistic (robust):             0.0276
                                        P-value                           0.8680
Time periods:                    11     Distribution:                  F(1,8854)
Avg Obs:                       805.09
Min Obs:                       515.00
Max Obs:                       1096.0

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
Intercept      0.0018     0.0031     0.5993     0.5490     -0.0042      0.0079
beta           0.0005     0.0027     0.1662     0.8680     -0.0049      0.0058
==============================================================================
```

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 0.0005 corresponds to a risk premium of 0.05% per month, whereas the (unweighted average) excess return for all firms in the sample is around –2% per year.

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 into the model. Let us start with a fixed effects model. To do so, we simply change the function **PooledOLS** to **PanelOLS** and add the string literals 'EntityEffects' into the formula. The dependent and independent variables remain the same as in the simple pooled regression.

Note that the linearmodels library offers many options to customise the model, including different standard error adjustments and weighting options. For now we will keep the default options. However, for future projects, the correct adjustment of standard errors is often a major consideration at the model selection stage. The final regression output should appear.

```
In [5]: # Panel Regression with Fixed Effects
        model = PanelOLS.from_formula('ret ~ 1 + beta + EntityEffects',data)
        res = model.fit()
        print(res)

                          PanelOLS Estimation Summary
================================================================================
```

```
Dep. Variable:                    ret   R-squared:                        0.0012
Estimator:                   PanelOLS   R-squared (Between):             -0.0119
No. Observations:                8856   R-squared (Within):               0.0012
Date:                Tue, Aug 14 2018   R-squared (Overall):             -0.0023
Time:                        14:12:14   Log-likelihood                 1.48e+04
Cov. Estimator:            Unadjusted
                                       F-statistic:                      8.3576
Entities:                        1734   P-value                           0.0039
Avg Obs:                       5.1073   Distribution:                  F(1,7121)
Min Obs:                       1.0000
Max Obs:                       11.000   F-statistic (robust):             8.3576
                                        P-value                           0.0039
Time periods:                      11   Distribution:                  F(1,7121)
Avg Obs:                       805.09
Min Obs:                       515.00
Max Obs:                       1096.0

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
Intercept      0.0155     0.0046     3.3827     0.0007      0.0065      0.0245
beta          -0.0119     0.0041    -2.8909     0.0039     -0.0200     -0.0038
==============================================================================

F-test for Poolability: 1.3111
P-value: 0.0000
Distribution: F(1733,7121)

Included effects: Entity
```

We can see that the estimate on the beta parameter is now negative and statistically significant, while the intercept is positive and statistically significant. We now estimate a random effects model. For this, we simply change the function again to **RandomEffects**. We leave all other specifications unchanged and execute these commands to generate the regression output.

```
In [6]: # Panel Regression with Random Effects
        model = RandomEffects.from_formula('ret ~ 1 + beta',data)
        res = model.fit()
        print(res)

                        RandomEffects Estimation Summary
================================================================================
Dep. Variable:                    ret   R-squared:                    7.135e-05
Estimator:              RandomEffects   R-squared (Between):             -0.0030
No. Observations:                8856   R-squared (Within):               0.0008
Date:                Tue, Aug 14 2018   R-squared (Overall):             -0.0019
Time:                        14:12:15   Log-likelihood                 1.412e+04
```

Cov. Estimator:	Unadjusted		
		F-statistic:	0.6318
Entities:	1734	P-value	0.4267
Avg Obs:	5.1073	Distribution:	F(1,8854)
Min Obs:	1.0000		
Max Obs:	11.000	F-statistic (robust):	2.8030
		P-value	0.0941
Time periods:	11	Distribution:	F(1,8854)
Avg Obs:	805.09		
Min Obs:	515.00		
Max Obs:	1096.0		

Parameter Estimates

	Parameter	Std. Err.	T-stat	P-value	Lower CI	Upper CI
Intercept	0.0063	0.0037	1.7208	0.0853	-0.0009	0.0136
beta	-0.0054	0.0032	-1.6742	0.0941	-0.0117	0.0009

The slope estimate is again of a different order of magnitude compared to both the pooled and the fixed effects regressions.

22 Limited dependent variable models

Reading: Brooks (2019, chapter 12)

Estimating limited dependent variable models in Python 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 preventing systematic biases in admissions decisions.[41]

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 the 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.[42] 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 a NoteBook. To check that all series are correctly imported, we can use the Pandas function **head** to visually examine the first five rows of imported data and the function **info** to get information on the characteristics of the dataset. Overall there should be 500 observations in the dataset for each series with no missing observations. Note that there are two variable names that have been renamed since the formula statement does not allow a variable name with spaces in between.

```
In [1]: import Pandas as pd
        import statsmodels.formula.api as smf
        import matplotlib.pyplot as plt

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        data = pd.read_excel(abspath + 'MSc_fail.xls')
```

[41] 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, we have deliberately retained as many fail observations in the sample as possible, which will bias the estimated failure rate upwards relative to the true rate.

[42] The exact identities of the countries involved are not given except that Country 8 is the UK!

```
data = data.rename(columns={'Work Experience':'WorkExperience',
                            'PG Degree':'PGDegree'})
data.head()
```

Out[1]:

	Age	Female	Fail	WorkExperience	English	Country Code	PGDegree	Agrade	\
0	22	1	0	0	0	10	0	0	
1	24	0	1	0	1	8	0	0	
2	28	0	0	0	1	10	0	0	
3	23	0	0	0	1	8	0	0	
4	23	0	0	1	1	8	0	0	

	BelowBGrade	Year2004	Year2005	Year2006	Year2007
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0

```
In [2]: data.info()
```

```
<class 'Pandas.core.frame.DataFrame'>
RangeIndex: 500 entries, 0 to 499
Data columns (total 13 columns):
Age               500 non-null int64
Female            500 non-null int64
Fail              500 non-null int64
WorkExperience    500 non-null int64
English           500 non-null int64
Country Code      500 non-null int64
PGDegree          500 non-null int64
Agrade            500 non-null int64
BelowBGrade       500 non-null int64
Year2004          500 non-null int64
Year2005          500 non-null int64
Year2006          500 non-null int64
Year2007          500 non-null int64
dtypes: int64(13)
memory usage: 50.9 KB
```

To begin with, suppose that we estimate a linear probability model of Fail on a constant, Age, English, Female, WorkExperience, A-Grade, Below-B-Grade, PG-Grade and the year dummies. This would be achieved simply by running a linear regression, using the function **ols**.

```
In [3]: # linear regression
        formula = 'Fail ~ Age + English + Female + WorkExperience + Agrade\
                        + BelowBGrade + PGDegree + Year2004 + Year2005 + Year2006\
                        + Year2007'
        mod = smf.ols(formula, data)
        res = mod.fit()
        print(res.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   Fail   R-squared:                       0.066
Model:                            OLS   Adj. R-squared:                  0.045
Method:                 Least Squares   F-statistic:                     3.148
Date:                Wed, 15 Aug 2018   Prob (F-statistic):           0.000396
Time:                        10:17:18   Log-Likelihood:                -153.89
No. Observations:                 500   AIC:                             331.8
Df Residuals:                     488   BIC:                             382.3
Df Model:                          11
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.1039      0.121      0.862      0.389      -0.133       0.341
Age            0.0013      0.004      0.305      0.761      -0.007       0.010
English       -0.0201      0.032     -0.637      0.525      -0.082       0.042
Female        -0.0294      0.035     -0.838      0.402      -0.098       0.039
WorkExperience -0.0620     0.031     -1.973      0.049      -0.124      -0.000
Agrade        -0.0807      0.038     -2.139      0.033      -0.155      -0.007
BelowBGrade    0.0926      0.050      1.844      0.066      -0.006       0.191
PGDegree       0.0287      0.047      0.605      0.546      -0.064       0.122
Year2004       0.0569      0.048      1.192      0.234      -0.037       0.151
Year2005      -0.0111      0.048     -0.230      0.819      -0.106       0.084
Year2006       0.1416      0.048      2.948      0.003       0.047       0.236
Year2007       0.0852      0.050      1.712      0.087      -0.013       0.183
==============================================================================
Omnibus:                      177.684   Durbin-Watson:                   2.032
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              412.578
Skew:                           1.922   Prob(JB):                     2.57e-90
Kurtosis:                       5.241   Cond. No.                         219.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

While this model has a number of very undesirable features, it would nonetheless provide a useful benchmark with which to compare the more appropriate models estimated below.

Next, we estimate a probit model and a logit model using the same dependent and independent variables as above. We begin with the logit model by calling it from **statsmodels.formula.api**. We keep the regression formula unchanged and apply While's robust standard errors as the correction for this specification. Using other parameters you can also change the optimisation method. However, we do not need to make any changes from the default, but simply execute the code cell. The output for the logit regression should appear as in the table below.

```
In [4]: # logit regression
        mod = smf.logit(formula, data)
        res = mod.fit(cov_type='HC1') # white robustness
        print(res.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.359433
         Iterations 7
                           Logit Regression Results
==============================================================================
Dep. Variable:                   Fail   No. Observations:                  500
Model:                          Logit   Df Residuals:                      488
Method:                           MLE   Df Model:                           11
Date:                Wed, 15 Aug 2018   Pseudo R-squ.:                0.08755
Time:                        10:17:18   Log-Likelihood:                -179.72
converged:                       True   LL-Null:                       -196.96
                                        LLR p-value:                 0.0003010
==============================================================================
                    coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept         -2.2564      1.220     -1.849      0.064      -4.648       0.135
Age                0.0110      0.046      0.240      0.810      -0.079       0.101
English           -0.1651      0.295     -0.560      0.576      -0.743       0.413
Female            -0.3339      0.360     -0.928      0.353      -1.039       0.371
WorkExperience    -0.5688      0.289     -1.968      0.049      -1.135      -0.002
Agrade            -1.0850      0.493     -2.203      0.028      -2.050      -0.120
BelowBGrade        0.5624      0.386      1.455      0.146      -0.195       1.320
PGDegree           0.2121      0.425      0.499      0.618      -0.621       1.046
Year2004           0.6532      0.484      1.351      0.177      -0.295       1.601
Year2005          -0.1838      0.559     -0.329      0.742      -1.280       0.912
Year2006           1.2466      0.469      2.657      0.008       0.327       2.166
Year2007           0.8504      0.482      1.765      0.078      -0.094       1.795
==============================================================================
```

Next we estimate the above model as a probit. We call the module **statsmodels.formula.api** but now select the built-in function **probit**. We input the same regression formula as in the logit case and again select robust standard errors. The output of the probit model is presented in the table on the following window. As can be seen, for both models the pseudo-R^2 values are quite small at just below 9%, although this is often the case for limited dependent variable models.

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%),[43] which makes it harder to fit a good model than if the proportion of passes and fails had been more evenly balanced.

```
In [5]: # probit regression
        mod = smf.probit(formula, data)
        res = mod.fit(cov_type='HC1')
        print(res.summary())
```

[43] Note that you can retrieve the number of observations for which 'Fail' takes the value 1 by using the command **np.where(data['Fail']==1,1,0)**.

```
Optimization terminated successfully.
         Current function value: 0.358913
         Iterations 6
                          Probit Regression Results
==============================================================================
Dep. Variable:                   Fail   No. Observations:                  500
Model:                         Probit   Df Residuals:                      488
Method:                           MLE   Df Model:                           11
Date:                Wed, 15 Aug 2018   Pseudo R-squ.:                0.08887
Time:                        10:17:18   Log-Likelihood:                -179.46
converged:                       True   LL-Null:                       -196.96
                                        LLR p-value:                 0.0002471
==============================================================================
                     coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept         -1.2872      0.610     -2.112      0.035      -2.482      -0.093
Age                0.0057      0.023      0.252      0.801      -0.039       0.050
English           -0.0938      0.156     -0.600      0.548      -0.400       0.212
Female            -0.1941      0.186     -1.042      0.297      -0.559       0.171
WorkExperience    -0.3182      0.151     -2.103      0.035      -0.615      -0.022
Agrade            -0.5388      0.231     -2.331      0.020      -0.992      -0.086
BelowBGrade        0.3418      0.219      1.559      0.119      -0.088       0.772
PGDegree           0.1330      0.226      0.589      0.556      -0.310       0.576
Year2004           0.3497      0.241      1.448      0.148      -0.124       0.823
Year2005          -0.1083      0.269     -0.403      0.687      -0.635       0.418
Year2006           0.6736      0.239      2.824      0.005       0.206       1.141
Year2007           0.4338      0.248      1.750      0.080      -0.052       0.920
==============================================================================
```

A further test on model adequacy is to produce a set of in-sample forecasts – in other words, to construct the fitted values. To do so, we access the predicted values from the regression result instance and plot this series by the function **plot**. We end up with the command **plt.show()** and the predicted series should appear in the output window (Figure 34).

```
In [6]: plt.figure(1)
        plt.plot(res.predict())
        plt.ylabel('Pr(Fail)')
        plt.xlabel('Seqnum')
        plt.show()
```

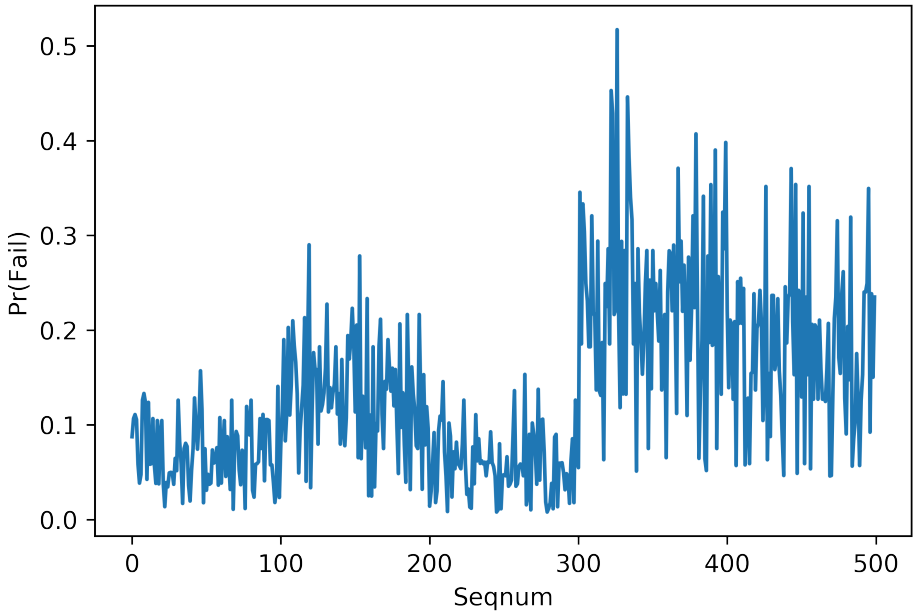


Figure 34: Graph of the Fitted Values from the Failure Probit Regression

The unconditional probability of failure for the sample of students we have is only 13.4% (i.e., only 67 out of 500 failed), so an observation should be classified as correctly fitted if either $y_i = 1$ and $\hat{y}_i > 0.134$ or $y_i = 0$ and $\hat{y}_i < 0.134$.

It is important to note that we cannot interpret the parameter estimates in the usual way (see the discussion in chapter 12 of Brooks (2019). In order to be able to do this, we need to calculate the marginal effects.

The regression instance has an in-built command that allows us to calculate marginal effects which can be accessed via **get_margeff().summary()**. Leaving all options to their default settings and executing the code cell, Jupyter should generate the table of marginal effects and corresponding statistics as shown on the following window.

```
In [7]: print(res.get_margeff().summary())

       Probit Marginal Effects
=====================================
Dep. Variable:                   Fail
Method:                          dydx
At:                           overall
================================================================================
                    dy/dx    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------------
Age                0.0011      0.004      0.252      0.801      -0.008       0.010
English           -0.0185      0.031     -0.601      0.548      -0.079       0.042
Female            -0.0382      0.037     -1.040      0.298      -0.110       0.034
WorkExperience    -0.0627      0.030     -2.101      0.036      -0.121      -0.004
```

```
Agrade            -0.1061      0.045     -2.342      0.019      -0.195      -0.017
BelowBGrade        0.0673      0.043      1.561      0.119      -0.017       0.152
PGDegree           0.0262      0.044      0.588      0.556      -0.061       0.113
Year2004           0.0689      0.048      1.445      0.149      -0.025       0.162
Year2005          -0.0213      0.053     -0.403      0.687      -0.125       0.082
Year2006           0.1326      0.047      2.840      0.005       0.041       0.224
Year2007           0.0854      0.049      1.757      0.079      -0.010       0.181
==============================================================================
```

We can repeat this exercise for the logit model using the same procedure as above. Note that we need to re-run the logit model first and then calculate marginal effects in the same way as described for the probit model. If done correctly, the table of marginal effects should resemble the following.

```
In [8]: mod = smf.logit(formula, data)
        res = mod.fit(cov_type='HC1')
        print(res.get_margeff().summary())

Optimization terminated successfully.
         Current function value: 0.359433
         Iterations 7
        Logit Marginal Effects
=====================================
Dep. Variable:                   Fail
Method:                          dydx
At:                           overall
==============================================================================
                    dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Age                0.0012      0.005      0.240      0.810      -0.008       0.011
English           -0.0178      0.032     -0.560      0.576      -0.080       0.044
Female            -0.0360      0.039     -0.925      0.355      -0.112       0.040
WorkExperience    -0.0613      0.031     -1.968      0.049      -0.122      -0.000
Agrade            -0.1169      0.053     -2.207      0.027      -0.221      -0.013
BelowBGrade        0.0606      0.042      1.455      0.146      -0.021       0.142
PGDegree           0.0228      0.046      0.498      0.618      -0.067       0.113
Year2004           0.0704      0.052      1.346      0.178      -0.032       0.173
Year2005          -0.0198      0.060     -0.329      0.742      -0.138       0.098
Year2006           0.1343      0.050      2.661      0.008       0.035       0.233
Year2007           0.0916      0.052      1.773      0.076      -0.010       0.193
==============================================================================
```

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.

23 Simulation methods

23.1 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
     $y_2 = y_1 + u_2$
     $y_3 = y_2 + u_3$
     ...
     $y_T = y_{T-1} + u_T$
2. Calculating the test statistic, τ.
3. Repeating 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. Ordering 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 Python code for conducting such a simulation is given below. The simulation framework considers a sample of 1,000 observations and DF 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.

```
In [1]: import NumPy as np
        import statsmodels.api as sm
        from statsmodels.tsa.tsatools import add_trend
```

Before turning to the custom function we designed to achieve this task, let us have a look at the set of commands that we want to perform first. As five parameters are used for input, we first need to convert the data type of these variables just in case that they are not fit to employ in the calculation. As can be seen in the code cell, we put **int** function for both **obs** and **alpha**. Besides converting the variables, an array, y, is created with all zero values, which will later be used to stored the fitted AR(1) values.

Next, the combined command with both **loops** and the **if** conditional statement is used to create a new variable y[1] that takes the value 0 for the first observation. Note that y[1] refers to the first observation, y[2] to the second observation, etc. The remaining values for y[t], from observations 2 to 1,000, will establish 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. It is very easy to construct such a series, the current value of the y[t] variable is referred to as the previous value of y[t-1] and the standard normal variate is added, where the coefficient of y is set to be one in this case.

Moving out of the loop, the next two commands **dy = np.diff(y)** and **y = y[:-1]** are used to generate first differences and lagged values of 'y', respectively.

Subsequently, the regression instance is constructed to generate the *t*-values. We set three **if** for three different circumstances where regression specifications include or exclude a constant or/and a trend. In particular, the first **if** command contains a DF regression of the first difference of y on the lagged value of y without constant and trend. For the second one, the constant is added so that the overall model contains an intercept but no linear trend. Finally, the last case is to generate the data from a model with both constant and trend. For each **if** command, the *t*-values will be returned depending upon the actual inputs are set for the 'const' and 'trend' term.

```
In [2]: # Simulate an AR(1) process with alpha = 1
        def simulate_AR(obs,alpha,w,const=False,trend=False):
            '''
            Simulate an AR(1) process.

            Parameters:
            -----------
            obs : the required number of observations.

            alpha: the coefficient of y.

            w: a set of error terms that follow a normal distribution.

            const: the boolean, whether to add a constant term in the formula.
                   default is false.

            trend: the boolean, whether to add a linear trend in the formula.
                   default is false.

            Returns:
            --------
            res.tvalues: the t-statistic of the regression instance under each
                         simulation.
            '''

            obs = int(obs)
            a = int(alpha)
            y = np.zeros(shape=(obs))

            for t in range(obs):
                if t == 0:
```

```
            y[t] = 0
        else:
            y[t] = a*y[t-1] + w[t]

    dy = np.diff(y)
    y = y[:-1]

    if (const is False) & (trend is False):
        res = sm.OLS(dy,y).fit()
        return res.tvalues

    if (const is True) & (trend is False):
        y = add_trend(y, trend='c')
        res = sm.OLS(dy,y).fit()
        return res.tvalues[0]

    if (const is True) & (trend is True):
        y = add_trend(y, trend='ct')
        res = sm.OLS(dy,y).fit()
        return res.tvalues[0]
```

To set out the simulation, the first few steps we need to conduct are merely preparation for the main simulation but are necessary to access the simulated critical values later on. Specifically, we first tell Python to create three separate NumPy arrays called 't1', 't2' and 't3', respectively, which will be used within the program to store the t-statistic for each simulation. Since 50,000 replications are used for simulation, these three arrays are generated with the same length (50,000). To conduct the simulation, a 'for' loop is required starting from zero to 50,000. Under each loop, the command **np.random.normal(size=1000)** sets the so-called random number seed. This is necessary 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. Next, calling the custom function **simulate_AR** that we defined will return the t-statistic depending on the constant and trend argument users set up. Apart from these two specifications, we also set the α coefficient for the 1,000 observations to be one, and we use error terms taking the random value from the normal distribution for the inputs.

Once finishing, the whitespace is killed to close the loop. We then print the critical values under each specification where 1%, 5% and 10% t-statistics are set up with or without the constant and trend accordingly. The function **percentile** provides us with the critical values for the three different models as it generates the 1st, 5th and 10th percentiles for each of the three variables 't1', 't2', 't3'.

```
In [3]: ## Model/Experiment ------------------------------

    t1 = np.zeros(shape=(50000))
    t2 = np.zeros(shape=(50000))
    t3 = np.zeros(shape=(50000))
    for i in range(50000):
        errors = np.random.normal(size=1000)
        t1[i] = simulate_AR(obs=1000, alpha=1, w=errors, const=False, trend=False)
        t2[i] = simulate_AR(obs=1000, alpha=1, w=errors, const=True, trend=False)
        t3[i] = simulate_AR(obs=1000, alpha=1, w=errors, const=True, trend=True)
```

```
        print('No const or trend: 1% {}'.format(np.percentile(t1,1)) )
        print('No const or trend: 5% {}'.format(np.percentile(t1,5)) )
        print('No const or trend: 10% {}'.format(np.percentile(t1,10)) )

        print('Const but no trend: 1% {}'.format(np.percentile(t2,1)) )
        print('Const but no trend: 5% {}'.format(np.percentile(t2,5)) )
        print('Const but no trend: 10% {}'.format(np.percentile(t2,10)) )

        print('Const and trend: 1% {}'.format(np.percentile(t3,1)) )
        print('Const and trend: 5% {}'.format(np.percentile(t3,5)) )
        print('Const and trend: 10% {}'.format(np.percentile(t3,10)) )
```

```
No const or trend: 1% -2.6014093678158994
No const or trend: 5% -1.960879813921448
No const or trend: 10% -1.625195012566009
Const but no trend: 1% -3.428109813365462
Const but no trend: 5% -2.8555719428447106
Const but no trend: 10% -2.562367147624574
Const and trend: 1% -3.954177929485786
Const and trend: 5% -3.411438023794303
Const and trend: 10% -3.124328787933637
```

The critical values obtained by running the above program, which are virtually identical to those found in the statistical tables at the end of Brooks (2019), are presented in the table above. 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 1000 observations is -1.96 in this simulation, and -1.95 in Fuller (2009).

23.2 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*.

Sample Python 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 FTSE100, 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, risk-free=6.24, dividend yield=2.42, 'today's' FTSE=6289.70, forward price=6405.35, implied volatility=26.52
Simulation 2: strike=5500, risk-free=6.24, dividend yield=2.42, 'today's' FTSE=6289.70, forward price=6405.35, 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 using Python is given as follows.

```
In [1]: import scipy as sp
        from NumPy import average, exp

        # parameters input
        S0 = 6289.70
        K = 6500 # excercise price
        T = 0.5 # maturity in years
        r = 0.0624 # risk-free rate
        d = 0.0242 # dividend yield
        sigma = 0.2652 # implied volatility
        n_simulation = 50000 # number of simulations
        n_steps = 125
        dt = T/n_steps
```

```
        call = sp.zeros([n_simulation],dtype=float)
        put = sp.zeros([n_simulation],dtype=float)

        for j in range(0, n_simulation):
            ST=S0
            total=0

            for i in range(0,int(n_steps)):
                e=sp.random.normal()
                ST=ST*sp.exp((r-d-0.5*sigma**2)*dt+sigma*sp.sqrt(dt)*e)
                total+=ST

            price_average=total/n_steps
            call[j]=max(price_average-K,0)*exp(-r*T)
            put[j]=max(K-price_average,0)*exp(-r*T)

        call_price=average(call)
        put_price=average(put)

        print('call price = ', round(call_price,2) )
        print('put price = ', round(put_price,2) )
```

```
call price =  203.772
put price =  351.562
```

As shown in the code cell, a list of variables is created by applying the parameters for simulation 1. Note that there are two variables **n_simulation** and **n_steps** that we set up; these indicate that we will be performing 50,000 repetitions of the set of commands (i.e., replications). Meanwhile, each replication will be performed for a set of 125 observations. A variable **dt** is generated by the T/n_steps equation, which splits the time to maturity (0.5 years) into 'N' discrete time periods. This variable will later be used in the geometric Brownian motion. To store the values of the call and put prices in each replication, we finally create two zero arrays before turning to the simulation. The command for creating this is **sp.zeros([n_simulation],dtype=float)**, where **n_simulation** is 50,000 and **zeros** is the built-in function from the **scipy** module.

The simulation will be carried out by a nested loop, or in other words, a loop within a loop. The first **for** loop is to iterate each of the 50,000 repetitions. To begin with, the underlying asset price 'ST' is set to be equal today's value, and the total value of asset prices for all steps is defined as zero. Next, the second **for** loop generates the path of the underlying asset. Since daily averaging is required, it is easiest to set i starting from 0 until 125 (the approximate number of trading days in half a year), so that each time period 'dt' represents one day. After that, the random normal draws are made by the function **sp.random.normal()**, which are then constructed into a series of futures prices of the underlying asset for the next 125. Intuitively, the essence of constructing the path of underlying asset prices is assuming that the underlying asset price follows a geometric Brownian motion under a risk-neutral measure, which is given by

$$dS = rf\text{ - }dySdt + \sigma Sdz \tag{15}$$

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

Table 2: Simulated Asian Option Price Values

Strike = 6500, IV = 26.52		Strike = 5500, IV = 34.33	
CALL	Price	*CALL*	Price
Analytical Approximation	203.45	Analytical Approximation	888.55
Monte Carlo Normal	203.77	Monte Carlo Normal	885.30
PUT	Price	*PUT*	Price
Analytical Approximation	348.7	Analytical Approximation	64.52
Monte Carlo Normal	351.56	Monte Carlo Normal	61.02

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

where u_t is a white noise error process.

Once all 125 observations have been created based on the formula above, we summarise the mean value of this series (i.e., the average price of the underlying asset over the lifetime of the option) which we capture by the command **price_average=total/n_steps**.

The following two sequences of commands construct the terminal payoffs for the call and put options, respectively. For the call, call[j] is set to 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, for each simulation. 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(-'r'*'T')'). These two values, i.e., the present value of the call and put options, are then printed to appear in the output window above

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 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 (2007).

In both cases, the simulated option prices are quite close to the analytical approximations. Some of the minor 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.

23.3 VaR estimation using bootstrapping

Reading: Brooks (2019, section 13.9)

The following Python 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 Excel workfile 'sp500_daily'. The code is presented as follows with explanations for each building block.

Again, we first import several modules and load in the data from the 'sp500_daily' Excel file. The custom function **LogDiff** is applied to obtain the log returns of the S&P500 index.

```
In [1]: import Pandas as pd
        import NumPy as np
        from arch import arch_model
        from NumPy import sqrt, exp, std, mean
        import statsmodels.api as sms

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        data = pd.read_excel(abspath + 'sp500_daily.xlsx',index_col=[0])

        def LogDiff(x):
            x_diff = np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data['rf'] = LogDiff(data['Close'])
        data = data.rename(columns={'Close':'sp500'})
        data = data.dropna()
```

The continuous log returns then become the dependent variable in the GARCH(1,1) model, i.e., one ARCH and one GARCH term, by using the **arch_model** function. The residuals can be accessed by calling the attribute of the GARCH result instance. Similarly, the square root of the conditional variance can be obtained by the attribute **conditional_volatility**. The equation 'res=resi/hsq' will construct a set of standardised residuals.[44]

```
In [2]: res = arch_model(data['rf'], vol='GARCH').fit()

        mu = res.params['mu']
        resi = res.resid

        hsq = res.conditional_volatility # square root of conditional variance
        sres = resi/hsq
```

```
Iteration:      1,   Func. Count:      6,   Neg. LLF: -58785.53002497787
Inequality constraints incompatible    (Exit mode 4)
            Current function value: -58785.530039173595
            Iterations: 1
            Function evaluations: 6
```

[44] The starting values for the GARCH model estimation may vary across different software/packages, which might later lead to a slight difference in the simulation results.

```
            Gradient evaluations: 1
```

Next, we set up a custom function **bootstrap_resample** where the the portion of the input sample will be randomly drawn. The input **X** would be the **sres** calculated in the previous block.

```
In [3]: def bootstrap_resample(X, n=None):
            """
            Bootstrap resample an array_like

            Parameters
            ----------
            X : array_like
              data to resample
            n : int, optional
              length of resampled array, equal to len(X) if n==None
            Results
            -------
            returns X_resamples
            """
            if n == None:
                n = len(X)

            resample_i = np.floor(np.random.rand(n)*len(X)).astype(int)
            X_resample = X[resample_i]
            return X_resample
```

Next follows the core of the program, which involves the bootstrap loop. 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, achieved by the function **bootstrap_resample**. Next, we construct the future path of the S&P500 return and price series over the ten-day holding period. Specifically, the first step to achieve this is to extend the sample period to include ten further observations, i.e., the command **forecasts=res.forecast(horizon=10)**. We then extend the conditional variance series by adding forecasts of this series. Once the conditional variance forecasts are created, we take the ten observations of the re-sampled standardised residuals and place them with the square root of the conditional variance to get forecast returns. This is achieved by the command **rtf = sqrt(forecasts_res).values*sres_b[:10].to_frame().values**.

Then, using these forecast return series, we can now construct the path of the S&P500 price series based on the formula

$$sp500_t^f = sp500_{t-1} e^{rt_t^f} \tag{17}$$

This formula is translated into Jupyter code as follows with a 'for' loop from 0 to 10. By doing so, we can create a series of lagged S&P500 prices and then generate the forecasts of the series over ten days. Finally, we obtain the minimum and maximum values of the S&P500 series over the ten days and store them in the variables 'minimum' and 'maximum'.

This set of commands is then repeated 10,000 times so that after the final repetition there will be 10,000 minimum and maximum values for the S&P500 prices.

```
In [4]: minimum = np.zeros(10000)
        maximum = np.zeros(10000)
```

```
for n in range(10000):
    sres_b = bootstrap_resample(sres)

    # calculate multiple-step-ahead forecast by built-in function
    forecasts = res.forecast(horizon=10)
    forecasts_res = forecasts.variance.dropna().T

    rtf = mu + sqrt(forecasts_res).values*sres_b[:10].to_frame().values

    sp500_f = np.zeros(10)
    for i in range(10):
        if i == 0:
            sp500_f[i] = data['sp500'][-1]*exp(rtf[i])
        else:
            sp500_f[i] = sp500_f[i-1]*exp(rtf[i])

    minimum[n] = min(sp500_f)
    maximum[n] = max(sp500_f)
```

The final block of commands generates the MCRR for a long and a short position in the S&P500. The first step is to construct the log returns for the maximum loss over the ten-day holding period, which for the long position is achieved by the command **long = np.log(minimum/2815.62)** and for the short position by **short = np.log(maximum/2815.62)**. We now want to find the fifth percentile of the empirical distribution of maximum losses for the long and short positions. Under the assumption that the both the **long** and **short** statistics are normally distributed across the replications, the MCRR can be calculated by the commands **1-exp(-1.645*std(long)+mean(long))** and **exp(1.645*std(short)+mean(short))-1**, respectively. The results generated by running the above program should be displayed in the Jupyter output window and should be: MCRR=0.03172729061586166 for the long position, and MCRR=0.03678997177441845 for the short position.

```
In [5]: long = np.log(minimum/2815.62)
        short = np.log(maximum/2815.62)

        print('mcrrl:{}'.format(1-exp(-1.645*std(long)+mean(long))))
        print('mcrrs:{}'.format(exp(1.645*std(short)+mean(short))-1))
```

```
mcrrl:0.03172729061586166
mcrrs:0.03678997177441845
```

These figures represent the minimum capital risk requirement for a long and short position, respectively, as a percentage of the initial value of the position for 95% coverage over a 10-day horizon. This means that, for example, approximately 3.2% 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 3.7%. This is as one would expect since the index had a positive drift over the sample period. Therefore, the index returns are not symmetric about zero as positive returns are slightly more likely than negative returns. Higher capital requirements are thus necessary for a short position since a loss is more likely than for a long position of the same magnitude.

24 The Fama-MacBeth procedure

Reading: Brooks (2019, section 14.2)

In this section we will perform the two-stage Fama and MacBeth (1973) procedure, which, together with related asset pricing tests, is described 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 Python 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-French 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.[45]

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 Python, this difference should not be consequential. The two data files used are 'monthlyfactors.csv' and 'vw_sizebm_25groups.csv'. 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 twenty-five value-weighted portfolios formed from a large universe of stocks, two-way sorted according to their sizes and book-to-market ratios.

Before conducting the a Fama-French or Carhart procedures using the methodology developed by Fama and MacBeth (1973), we need to import the two csv data files into Python. The data in both cases run from October 1980 to December 2012, making a total of 387 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 split the Python code into several parts. The first step is to import and slice the subsample data, and 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 a loop to transform the original series to their excess return counterparts.

```
In [1]: import Pandas as pd
        import statsmodels.api as sm
        import NumPy as np

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                          'QMF Book/book Ran/data files new/Book4e_data/'
        data1 = pd.read_csv(abspath + 'monthlyfactors.csv',index_col=[0])
        data2 = pd.read_csv(abspath + 'vw_sizebm_25groups.csv',index_col=[0])

        # subsample
        data1 = data1['1980m10':'2010m12']*100
        data2 = data2['1980m10':'2010m12']*100
```

[45] http://businessschool.exeter.ac.uk/research/areas/centres/xfi/research/famafrench/files.

```
        # excess return
        for each in data2.columns:
            data2[each] = data2[each]-data1['rf']
```

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 4-factor model separately for each of the twenty-five portfolios. A Carhart 4-factor model regresses a portfolio return on the excess market return ('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 twenty-five portfolios, we can set this first stage up as a loop.

In particular, the line of command **y = data2[[each]]** specifies that for each of the 25 variables listed in the DataFrame **data2**, Python will then perform an OLS regression of the portfolio's excess return (indicated by the 'y' which is to be replaced by the particular portfolio return listed in the loop) on the four factors. Note that we restrict the sample for this model to the period October 1980 to December 2010 (**data1 = data1['1980m10':'2010m12']100**) instead of using the whole sample.

We need to store the estimates from these regressions into separate series for each parameter. This is achieved by the command **betas.append(res)**. In particular, **betas** is a newly-created list in which the beta coefficients on the four factors plus the constant are saved. Thus, for each of the twenty-five portfolios, we will generate four beta estimates that will then be posted to the list **betas**. After the loop has ended, the **betas** will contain 25 observations for the five variables 'const', 'smb', 'hml', 'umd' and 'rmrf'.

To illustrate this further, the loop starts off with the excess return on portfolio 'SL' (the first item in the **data2.columns**) and the coefficient estimates from the 4-factor model ('const', 'smb', 'hml', 'umd' and 'rmrf') will be transferred to the **betas**.

Then Python moves on to the second variable in the **data2.columns**, i.e., the excess return on portfolio 'S2', and will perform the 4-factor model and save the coefficient estimates in **betas**. Python will continue with this procedure until it has performed the set of commands for the last variable in the list, i.e., 'BH'. The beta coefficients for the regressions using 'BH' as the dependent variable will be the 25th entry in the **betas**.

Finally, the line of command **betas = pd.concat(betas, axis=1)** brings all of the separate estimates into an integrated DataFrame.

```
In [2]: # first stage
        x = data1[['smb','hml','umd','rmrf']]
        x = sm.add_constant(x)
        betas = []
        for each in data2.columns:
            y = data2[[each]]
            res = sm.OLS(y,x).fit().params
            res = res.rename(each)
            betas.append(res)

        betas = pd.concat(betas, axis=1)
```

So now we have 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 (i.e. 'const', 'smb', 'hml', 'umd' and 'rmrf') will show how sensitive the returns on that portfolio are to the corresponding factors, and the intercepts ('const') will be the Jensen's alpha estimates.

These intercept estimates stored in 'const' 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 all of the figures given from the Python output by 100 to make them on the same scale. If the 4-factor model is a good one, we should find that all of these alphas are statistically insignificant. We could test this individually if we wished by adding an additional line of code in the loop to save the t-ratio in the regressions.

The second stage of the Fama-MacBeth procedure is to run a separate cross-sectional regression for each point in time. An easy way to do this is to, effectively, rearrange/transpose the data so that each column represents one month and the rows are the excess returns of one of the 25 portfolios. In Python this can be easily achieved by the function **transpose()**, which transposes the data, changing variables into observations and observations into variables. The new variable **x** then stores the transposed data containing the original variable names.

Now Python has new variable names for the observations. For example, all data points in the first row (of the previous dataset) corresponding to October 1980 are now stored in the first column; all data points in the second row corresponding to November 1980 are now stored in the second column; and so on. Thus, the first column contains the excess returns of all 25 portfolios for October 1980 and the last column contains the excess returns of the 25 portfolios for December 2010.

Additionally, the constant term for the **x** needs to be included manually since we are applying the **OLS** function instead of the **formula** function. Note that the latter is able to automatically add constant terms.

We are now in a position to run the second-stage cross-sectional regressions corresponding to the following equation.[46]

$$R_i = \alpha + \lambda_M \beta_{i,M} + \lambda_S \beta_{i,S} + \lambda_V \beta_{i,V} + \lambda_U \beta_{i,U} + e_i \tag{18}$$

Again, it is more efficient to run this set of regressions in a loop. In particular, we regress each of the columns (i.e., '1980m10', '1980m11', '1980m12',..,'2010m12') representing the excess returns of the 25 portfolios for a particular month on the beta estimates from the first stage (i.e., 'const', 'smb', 'hml', 'umd' and 'rmrf'). Then we store the coefficients on these factor exposures in an another list object **lambdas**. 'const' will contain all intercepts from the second-stage regressions, 'rmrf' will contain all parameter estimates on the market risk premium betas, and so on.

Likewise, we store all of these estimates under each specification in the list **lambdas** and convert these separated series into an integrated DataFrame.

```
In [3]: # second stage
        x = betas.transpose()[['smb','hml','umd','rmrf']]
        x = sm.add_constant(x)
        lambdas = []
        for each in data2.transpose().columns:
            y = data2.transpose()[[each]]
            res = sm.OLS(y,x).fit().params
            res = res.rename(each)
            lambdas.append(res)

        lambdas = pd.concat(lambdas, axis=1)
```

The final stage of the Fama-MacBeth procedure is to estimate the averages and standard deviations of these estimates using something equivalent to the following equations, respectively for each parameter. The average value over t of $\hat{\lambda}_{j,t}$ can be calculated as follows.

[46] See Brooks (2019) for an explanation of this equation.

$$\hat{\lambda}_j = \frac{1}{T_{FMB}} \sum_{t=1}^{T_{FMB}} \hat{\lambda}_{j,t}, j = 1, 2, 3, 4, ... \tag{19}$$

To do this we first create a new variable **lambdas1** containing the mean of the estimates (**lambdas1 = lambdas.mean(axis=1)**) and a variable **lambdas2** containing the t-ratio of the estimates (**lambdas2 = lambdas.std(axis=1)**). Thus **lambdas1** will contain the mean of the cross-sectional intercept estimates, and the corresponding t-ratio will be stored in **lambdas3** given the command **lambdas3 = np.sqrt(lambdas.shape[1])lambdas1/lambdas2**.

```
In [4]: # final stage
        lambdas1 = lambdas.mean(axis=1)
        lambdas2 = lambdas.std(axis=1)
        lambdas3 = np.sqrt(lambdas.shape[1])*lambdas1/lambdas2
```

Finally we tell Python to display these values in order to inspect the results of the Fama-MacBeth procedure. The lambda parameter estimates should be comparable with 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. The parameter estimates obtained from this simulation and their corresponding t-ratios are given in the table below. The latter do not incorporate 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.

```
In [5]: output = pd.concat([lambdas1,lambdas3],join='inner', axis=1)
        output = output.rename(columns={0:'Estimates',
                                        1:'t-ratio'})
        output
```

```
Out[5]:        Estimates   t-ratio
        const   0.496870  1.327892
        smb     0.085269  0.511656
        hml     0.394470  2.083069
        umd     0.411830  0.777792
        rmrf    0.069129  0.151575
```

25 Using extreme value theory for VaR calculation

Reading: Brooks (2019, section 14.3

In this section, we are going to implement extreme value theory (EVT) in order to calculate VaR. The following Python code can be used to calculate VaR for both the delta normal and the historical distribution on a daily basis using & P500 data. The data can be retrieved from the Excel workfile 'sp500_daily' and stored in the Python DataFrame **data**, details of which can be found in the first command block. After computing the daily log returns of the S&P500 index, we obtain the parameters (i.e., the mean and standard deviation) later used to describe the distribution. The parameters can easily be calculated by the commands **mean = np.mean(data['ret'])** and **std_dev = np.std(data['ret'])**.

```
In [1]: import Pandas as pd
        import NumPy as np
        import matplotlib.pyplot as plt
        import matplotlib.mlab as mlab
        from scipy.stats import norm

        abspath = 'C:/Users/tao24/OneDrive - University of Reading/PhD/' \
                            'QMF Book/book Ran/data files new/Book4e_data/'
        data = pd.read_excel(abspath + 'sp500_daily.xlsx',index_col=[0])

        def LogDiff(x):
            x_diff = np.log(x/x.shift(1))
            x_diff = x_diff.dropna()
            return x_diff

        data['ret'] = LogDiff(data['Close'])
        data = data.rename(columns={'Close':'sp500'})
        data = data.dropna()

        mean = np.mean(data['ret'])
        std_dev = np.std(data['ret'])
```

Next, we plot both a histogram and the probability density function (pdf) of the standard normal distribution given the calculated parameters (**mean**, **std_dev**). As can be seen in the output window (Figure 35), both distributions are very close to each other given that the input sample size is quite large.

```
In [10]: fig = plt.figure(1, dpi=600)
         ax1 = fig.add_subplot(111)
         ax1.hist(data['ret'],100,edgecolor='black',linewidth=1.2)
         ax1.set_ylabel('density')

         x = np.linspace(mean - 5*std_dev, mean + 5*std_dev, 100)
         ax2 = ax1.twinx()
         ax2.plot(x,mlab.normpdf(x,mean,std_dev),"r")
         ax2.set_ylabel('pdf of the normal distribution')
         plt.legend()
         plt.show()
```

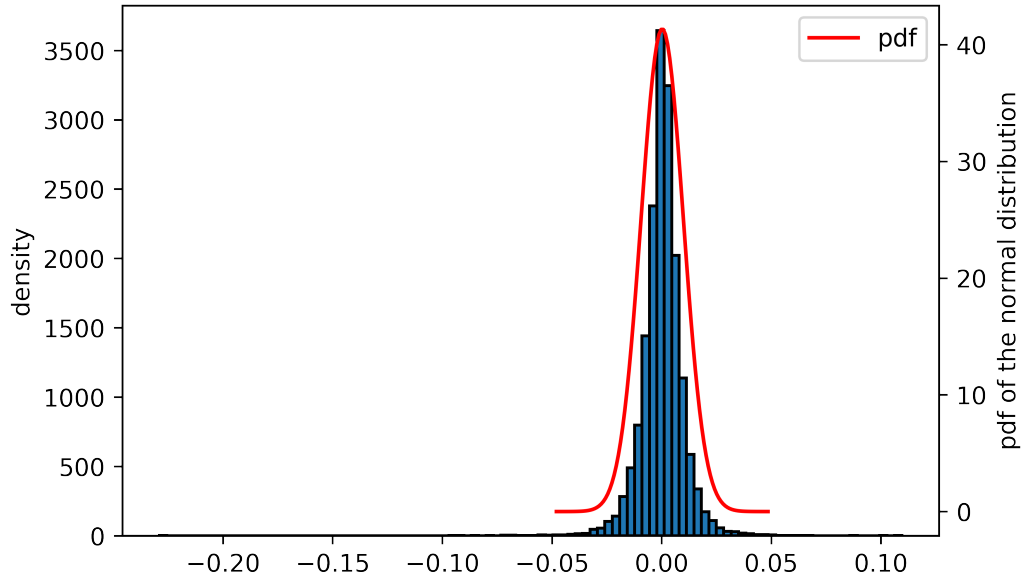


Figure 35: Histogram and the Probability Density Function of the Standard Normal Distribution

There are several ways to compute value at risk. The first calculation assumes that the return distribution is normal, and then we can simply take the critical value from the normal distribution at the α significance level multiplied by the standard deviation σ of the sample and subtracting the mean value.

$$VaR_{normal} = \sigma * Z_{\alpha} - \mu \tag{20}$$

In Python, the calculation can easily be done by applying the **ppf** function (i.e., the Percent Point function, which is the inverse of cdf – percentiles) from the **scipy.stats.norm** module. The specific commands are provided as follows. For VaR at the 90% confidence level, the input quantile is simply 1-0.9. A 95% VaR would be 1-0.95, and so on.

```
In [11]: # normal distribution
         VaR_90 = norm.ppf(1-0.9, mean, std_dev)
         VaR_95 = norm.ppf(1-0.95, mean, std_dev)
         VaR_99 = norm.ppf(1-0.99, mean, std_dev)

         print('90%:{}'.format(VaR_90) )
         print('95%:{}'.format(VaR_95) )
         print('99%:{}'.format(VaR_99) )
```

```
90%:-0.012067156368399046
95%:-0.01557235218232069
99%:-0.022147516586450203
```

Alternatively, VaR can be calculated by sorting the portfolio returns and then simply selecting the quantile from the empirical distribution of ordered historical returns. In this setting, the non-parametric VaR estimation would be under no assumptions, which is therefore fairly simple. Specifically, the Pandas module provides the quantile function where the corresponding VaR can be obtained by finding the appropriate quantiles, as shown in the following commands.

```
In [12]: # historical distribution
         VaR_90 = data['ret'].quantile(0.1)
         VaR_95 = data['ret'].quantile(0.05)
         VaR_99 = data['ret'].quantile(0.01)

         print('90%:{}'.format(VaR_90) )
         print('95%:{}'.format(VaR_95) )
         print('99%:{}'.format(VaR_99) )
```

```
90%:-0.009886539705571379
95%:-0.014438124291662094
99%:-0.025965714775369006
```

Finally, let us spend a few minutes comparing the VaR calculated by the different methods. Clearly, the historical 99% VaR is larger in absolute value than the corresponding normal VaR (-0.026 vs -0.022). The rationale behind this finding is that real financial data tend to have a distribution with fatter tails than the assumed standard normal distribution.

The Hill estimator for extreme value theory

In this subsection we will examine the tail of the distribution to estimate the VaR by implementing the Hill estimator of the shape parameter ξ of an extreme value distribution. The code to achieve this is presented as follows.

```
In [5]: threshold = -0.025
        alpha = 0.01

        data1 = data[data['ret']<=threshold]
        data1 = data1.sort_values(['ret'])
        data1['k'] = data1.iloc[0]['ret']

        xi = (np.log(-data1['k']) - np.log(-data1['ret']) ).sum()*(1/(data1.shape[0]-1))
        VaR_hill = data1['ret']*(data.shape[0]/data1.shape[0]*alpha)**(-xi)
        VaR_hill = VaR_hill[1:]
```

The threshold is set to be –0.025, which determines the number of observations identified as being in the tail. Note that this value is subjective and readers can choose an appropriate threshold at their own discretion. α is equal to 0.01 and corresponds to a 99% confidence level. To calculate the tail-index ξ using the Hill estimator, we order the data over the threshold from the largest value (lowest log returns) to the smallest value (largest log returns). By following the Hill estimator formula below, we then obtain the ξ and calculate the VaR – see Brooks (2019) for further details.

$$\hat{\xi} = \frac{1}{k-1}\Sigma_{i=1}^{k-1}[ln(\tilde{y}_i) - ln(\tilde{y}_k)] \tag{21}$$

```
In [6]: plt.figure(2, dpi=600)
        x = [i for i in range(1,VaR_hill.shape[0]+1)]
        plt.scatter(x,VaR_hill)
        plt.xlabel('n')
        plt.ylabel('VaR')
        plt.show()
```

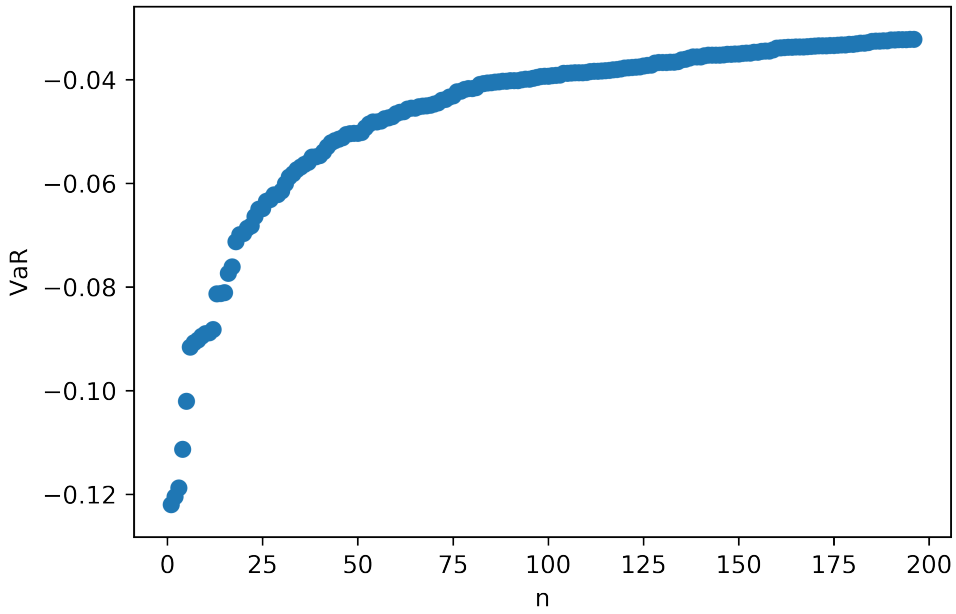


Figure 36: Hill Plot

Looking at the Hill plot (see Figure 36), we can see that the estimated VaR starts from a very negative value (–0.12) and gradually converges on –0.032, which suggests that the expected maximum is 3.2%

Parameter estimation for extreme value theory

The three parameters ξ, μ, σ used to define an extreme value distribution can be estimated by maximum likelihood. To do so, we need to set up a log-likelihood function with a certain distribution and find the parameter values to maximise it. The Python code below demonstrates this in detail. To be consistent with the previous section, the same values are set up for both alpha and the threshold. All observations exceeding the threshold $\tilde{y}\_t$ are obtained and stored in the variable **rets**. Next, we set up initial values for the distribution's parameter ξ and σ and we allow them to be optimised in the later maximum likelihood estimation.

```
In [7]: import NumPy as np
        from scipy.optimize import minimize

        alpha = 0.01
```

```
threshold = -0.025
y = threshold - data[data['ret']<=threshold]
y = y.sort_values(['ret'])
rets = y['ret'].values

shape = 0.2
scale = 1
x0 = (shape, scale)
```

We then define a custom function where the natural log of the probability density function is added up over the tail observations. By minimising the log-likelihood function by the function **minimize** from the module **scipy.optimize**, we are able to obtain optimised parameters, which are 0.38803644 and 0.0075239 for the distribution's shape and scale in this case.[47]

```
In [8]: def fun(params,rets):
            F = lambda x: 1/params[1]*(1 + params[0] * x/params[1])**(-1 - 1/params[0])
            result = -sum([np.log(d) for d in [F(x) for x in rets] ])
            return result

        result = minimize(fun, x0, args=rets, method='Nelder-Mead')
        print (result)

 final_simplex: (array([[ 0.38803644,  0.0075239 ],
       [ 0.38811469,  0.00752272],
       [ 0.38812762,  0.00752404]]), array([-689.8224412 , -689.82244075, -689.82244058]))
           fun: -689.82244119932989
       message: 'Optimization terminated successfully.'
          nfev: 99
           nit: 51
        status: 0
       success: True
             x: array([ 0.38803644,  0.0075239 ])
```

Finally, we calculate the VaR under maximum likelihood, which suggests the expected maximum loss is 2.5%.

```
In [9]: shape = result.x[0]
        scale = result.x[1]

        VaR_mle = threshold + scale/shape*((data.shape[0]/y.shape[0]*alpha)**(shape)-1)
        print('VaR_mle:{}'.format(VaR_mle) )

VaR_mle:-0.025975570332890704
```

[47] Note that, effectively, we are minimising the negative of the log-likelihood function, which is exactly equivalent to maximising the log-likelihood function but is simpler to implement since it means that we are able to use the **minimise** function in Python.

References

Brooks, C. (2019). *Introductory Econometrics for Finance*, volume 4th edition. Cambridge University Press, Cambridge, UK.

Carhart, M. (1997). On persistence in mutual fund performance. *Journal of Finance*, 52(1):57–82.

Durbin, J. and Watson, G. S. (1951). Testing for serial correlation in least squares regression. ii. *Biometrika*, 38(1/2):159–177.

Engle, R. (1982). Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica*, 50(4):987–1007.

Fama, E. F. and MacBeth, J. D. (1973). Risk, return, and equilibrium: Empirical tests. *Journal of Political Economy*, 81(3):607–636.

Fuller, W. (2009). *Introduction to Statistical Time Series*, volume 428. John Wiley & 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 & Accounting*, 40(1-2):172–214.

Haug, E. G. (2007). *The Complete Guide to Option Pricing Formulas*, volume 2. McGraw-Hill, New York.

Nelson, D. (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.