Community Experience Distilled

Mastering Python for Finance

Understand, design, and implement state-of-the-art mathematical and statistical applications used in finance with Python

James Ma Weiming

[PACKT] PUBLISHING open source* community experience distilled

Mastering Python for Finance

Understand, design, and implement state-of-the-art mathematical and statistical applications used in finance with Python

James Ma Weiming

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Mastering Python for Finance

First published: April 2015

Production reference: 1240415

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-451-6

www.packtpub.com

Credits

Author
James Ma Weiming

Reviewers
Namit Kewat
Marco Marchioro
Jiri Pik
Steven E. Sommer, MD, MBA

Commissioning Editor
Usha Iyer

Acquisition Editor
Usha Iyer

Content Development Editor
Susmita Sabat

Technical Editor
Prajakta Mhatre

Copy Editor
Rashmi Sawant

Project Coordinator
Milton Dsouza

Proofreaders
Stephen Copestake
Safis Editing
Paul Hindle

Indexer
Hemangini Bari

Graphics
Sheetal Aute
Valentina D'silva
Disha Haria
Abhinash Sahu

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

James Ma Weiming works with high-frequency, low-latency trading systems, writing his own programs and tools, most of which are open sourced. He is currently supporting veteran traders in the, trading pits of the Chicago Board of Trade devising strategies to game the market. He graduated from the Stuart School of Business at Illinois Institute of Technology with a master of science degree in finance.

He started his career in Singapore after receiving his bachelor's degree in computer engineering from Nanyang Technological University and diploma in information technology from Nanyang Polytechnic. During his career, he has worked in treasury operations handling foreign exchange and fixed income products. He also developed mobile applications with a company operating a funds and investments distribution platform.

This book was made possible by the fabulous team at Packt Publishing, especially Usha Iyer and Susmita Sabat. I am also eminently grateful to all the reviewers for their comments and to my immediate family for their encouragement and good cheer.

I'd like to thank Milt Robinson, Brian Hickman, and Frank for their mentorship on the trading floor.

About the Reviewers

Namit Kewat is a financial analyst and XBRL expert. He uses Python for his requirements related to financial reporting, from extracting data to its validation, and from recording to reporting. In his spare time, he enjoys working on web projects, machine learning experiments on SEC/HMRC XBRL financial data, and spending time with his family.

Marco Marchioro is the CEO of Quant Island, a Singapore-based consultancy firm specialized in quantitative risk models for asset management and energy finance. He has 15 years of experience in quantitative financial risk management, where his areas of expertise range from quantitative risk modeling and agile software development, to risk training. As a founding partner of RiskMap, he was one of the three creators of QuantLib, a widely-used open source library for financial modeling. He has extensive experience in quantitative finance, where he is well-versed with the end-to-end process of developing financial software. Prior to moving to Singapore, he held various senior roles in StatPro, covering the risk-management software development cycle. As the head of the quantitative research team, he was responsible for creating original risk models that have been successively and quickly implemented in an agile software environment. From 2010 to 2014, he held the position of an adjunct professor at the University of Milano-Bicocca, where he taught complex derivatives to a highly-ranked graduate class.

Jiri Pik is a finance and business intelligence consultant, working with major investment banks, hedge funds, and other financial players. He has architected and delivered breakthrough trading, portfolio and risk management systems, and decision support systems across a number of industries.

Jiri's consulting firm, WIXESYS, provides its clients with certified expertise, judgment, and execution at the speed of light. WIXESYS' power tools include revolutionary Excel and Outlook add-ons, available at `http://spearian.com`.

Steven E. Sommer, MD, MBA is a physician who has practiced critical care medicine for over 24 years. He is the chief investment officer for a small hedge fund, where he has employed portfolio optimization models based on volatility, modern portfolio theory, and market regime to drive asset selection and market exposure decisions. He has extensively employed R and Python to leverage big data in the development of his investment models.

Dr. Sommer holds a BA degree from Lafayette College, where he graduated Magna Cum Laude with honors in chemistry, an MD degree from Drexel University School of Medicine, where he graduated with distinction in medicine, an MBA degree from the University of Virginia's Darden School of Business, and a certificate in computational finance with distinction from the Georgia Institute of Technology.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Python is widely practiced in various sectors of finance, such as banking, investment management, insurance, and even real estate, for building tools that help in financial modeling, risk management, and trading. Even big financial corporations embrace Python to build their infrastructure for position management, pricing, risk management, and trading systems.

Throughout this book, theories from academic financial studies will be introduced, accompanied by their mathematical concepts to help you understand their uses in practical situations. You will see how Python is applied to classical pricing models, linearity, and nonlinearity of finance, numerical procedures, and interest rate models, that form the foundations of complex financial models. You will learn about the root-finding methods and finite difference pricing for developing an implied volatility curve with options.

With the advent of advanced computing technologies, methods for the storing and handling of massive amounts of data have to be considered. Hadoop is a popular tool in big data. You will be introduced to the inner workings of Hadoop and its integration with Python to derive analytical insights on financial data. You will also understand how Python supports the use of NoSQL for storing non-structured data.

Many brokerage firms are beginning to offer APIs to customers to trade using their own customized trading software. Using Python, you will learn how to connect to a broker API, retrieve market data, generate trading signals, and send orders to the exchange. The implementation of the mean-reverting and trend-following trading strategies will be covered. Risk management, position tracking, and backtesting techniques will be discussed to help you manage the performance of your trading strategies.

The use of Microsoft Excel is pervasive in the financial industry, from bond trading to back-office operations. You will be taught how to create numerical pricing Component Object Model (COM) servers in Python that will enable your spreadsheets to compute and update model values on the fly.

What this book covers

Chapter 1, *Python for Financial Applications*, explores the aspects of Python in judging its suitability as a programming language in finance. The IPython Notebook is introduced as a beneficial tool to visualize data and to perform scientific computing.

Chapter 2, *The Importance of Linearity in Finance*, uses Python to solve systems of linear equations, perform integer programming, and apply matrix algebra to linear optimization of portfolio allocation.

Chapter 3, *Nonlinearity in Finance*, discusses the nonlinear models in finance and root-finding methods using Python.

Chapter 4, *Numerical Procedures*, explores trees, lattices, and finite differencing schemes for valuation of options.

Chapter 5, *Interest Rates and Derivatives*, discusses the bootstrapping process of the yield curve and covers some short rate models for pricing the interest rate derivatives with Python.

Chapter 6, *Interactive Financial Analytics with Python and VSTOXX*, discusses the volatility indexes. We will perform analytics on EURO STOXX 50 Index and VSTOXX data, and replicate the main index using options prices of the sub-indexes.

Chapter 7, *Big Data with Python*, walks you through the uses of Hadoop for big data and covers how to use Python to perform MapReduce operations. Data storage with NoSQL will also be covered.

Chapter 8, *Algorithmic Trading*, discusses a step-by-step approach to develop a mean-reverting and trend-following live trading infrastructure using Python and the API of a broker. Value-at-risk (VaR) for risk management will also be covered.

Chapter 9, *Backtesting*, discusses how to design and implement an event-driven backtesting system and helps you visualize the performance of our simulated trading strategy.

Chapter 10, *Excel with Python*, discusses how to build a Component Object Model (COM) server and client interface to communicate with Excel and to perform numerical pricing on the call and put options on the fly.

What you need for this book

In this book, the following software will be required:

- The operating systems are as follows:
  - Any operating system with Python 2.7 or higher installed
  - Microsoft Windows XP or superior for *Chapter 10*, *Excel with Python*
  - A 64-bit host operating system with 4 GB of RAM for *Chapter 7*, *Big Data with Python*
- One of the following Python distribution packages that include Python, SciPy, pandas, IPython, and Matplotlib modules, which will be used throughout this book:
  - Anaconda 2.1 or higher from Continuum Analytics at `https://store.continuum.io/cshop/anaconda/`
  - Canopy 1.5 or higher from Enthought at `https://store.enthought.com/downloads/`
- Additional required Python modules are as follows:
  - Statsmodels at `http://statsmodels.sourceforge.net/`
  - PuLP for *Chapter 2*, *The Importance of Linearity in Finance* at `https://github.com/coin-or/pulp`
  - lxml for *Chapter 6*, *Interactive Financial Analytics with Python and VSTOXX* at `http://lxml.de/`
  - PyMongo 2.7 for *Chapter 7*, *Big Data with Python* at `https://pypi.python.org/pypi/pymongo/`
  - IbPy for *Chapter 8*, *Algorithmic Trading* at `https://github.com/blampe/IbPy`
  - oandapy for *Chapter 8*, *Algorithmic Trading* at `https://github.com/oanda/oandapy`
  - python-requests for *Chapter 8*, *Algorithmic Trading* at `https://pypi.python.org/pypi/requests/`
  - PyWin32 for *Chapter 10*, *Excel with Python* at `http://sourceforge.net/projects/pywin32/files/`
- Optional Python modules are as follows:
  - pip 6.0 to install Python packages automatically, at `https://pypi.python.org/pypi/pip`

- The required softwares are as follows:
    - Mozilla Firefox at `https://www.mozilla.org/en-US/firefox/new/`
    - MongoDB 2.6 for *Chapter 7*, *Big Data with Python* at `http://www.mongodb.org/downloads`
    - VirtualBox 4.3 for *Chapter 7*, *Big Data with Python* at `https://www.virtualbox.org/wiki/Downloads`
    - Cloudera QuickStart VM with CDH (Cloudera Distribution Including Apache Hadoop) for *Chapter 7*, *Big Data with Python* at `http://www.cloudera.com/content/cloudera/en/downloads/quickstart_vms.html`
    - Interactive Brokers (IB) Trader Workstation (TWS) for *Chapter 8*, *Algorithmic Trading* at `https://www.interactivebrokers.com/en/index.php?f=1537`
    - Oracle Java 7 to run IB TWS and OANDA fxTrade platform for *Chapter 8*, *Algorithmic Trading*.
    - Microsoft Office Excel 2010 or higher with developer and macros enabled for *Chapter 10*, *Excel with Python*.

Who this book is for

This book is geared toward students and programmers developing financial applications, consultants offering financial services, financial analysts, and quants who would like to master finance by harnessing Python's strengths in data visualization, interactive analytics, and scientific computing. An intermediate level of Python knowledge and financial concepts is expected. Beginners will receive an introductory background before jumping into the technical process of each chapter.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `price` function of the `BinomialEuropeanOption` class is a public method that is the entry point for all the instances of this class."

A block of code is set as follows:

```
def _traverse_tree_(self, payoffs):
    # Starting from the time the option expires, traverse
    # backwards and calculate discounted payoffs at each node
    for i in range(self.N):
        payoffs = (payoffs[:-1] * self.qu +
                   payoffs[1:] * self.qd) * self.df
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
Set BinCRRTree = CreateObject("BinomialCRRCOMServer.Pricer")
answer = BinCRRTree.pricer(S0, K, r, T, N, sigma, isCall, _
    dividend, True)
```

Any command-line input or output is written as follows:

```
>>> from FDCnEu import FDCnEu
>>> option = FDCnEu(50, 50, 0.1, 5./12., 0.4, 100, 100,
...                 100, False)
>>> print option.price()
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "We can compile the code by selecting **Debug** from the toolbar menu and clicking on **Compile VBAProject**:"

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Python for Financial Applications

In this introductory chapter, we will explore the aspects of Python in order to judge its suitability as a programming language in finance. Notably, Python is widely practiced in various financial sectors, such as banking, investment management, insurance, and even in real estate for building tools that help in financial modeling, risk management, and trading. To help you get the most from the multitude of features that Python has to offer, we will introduce the IPython Notebook as a beneficial tool to help you visualize data and to perform scientific computing for presentation to end users.

In this chapter, we will cover the following topics:

- Benefits of Python over other programming languages for financial studies
- Features of Python for financial applications
- Implementing object-oriented design and functional design in Python
- Overview of IPython
- Getting IPython and IPython Notebook started
- Creating and saving notebook documents
- Various formats to export a notebook document
- Notebook document user interface
- Inserting Markdown language into a notebook document
- Performing calculations in Python in a notebook document
- Creating plots in a notebook document
- Various ways of displaying mathematical equations in a notebook document
- Inserting images and videos into a notebook document
- Working with HTML and pandas DataFrame in a notebook document

Is Python for me?

Today's financial programmers have a diverse choice of programming languages in implementing robust software solutions, ranging from C, Java, R, and MATLAB. However, each programming language was designed differently to accomplish specific tasks. Their inner workings, behavior, syntax, and performance affect the results of every user differently.

In this book, we will focus exclusively on the use of Python for analytical and quantitative finance. Originally intended for scientific computations, the Python programming language saw an increasingly widespread use in financial operations. In particular, pandas, a software library written for the Python programming language, was open sourced by an employee of AQR Capital Management to offer high-performance financial data management and quantitative analysis.

Even big financial corporations embrace Python to architect their infrastructure. Bank of America's Quartz platform uses Python for position management, pricing, and risk management. JP Morgan's Athena platform, a cross-market risk management and trading system, uses Python for flexibility in combination with C++ and Java.

The application of Python in finance is vast, and in this book, we will cover the fundamental topics in creating financial applications, such as portfolio optimization, numerical pricing, interactive analytics, big data with Hadoop, and more.

Here are some considerations on why you might use Python for your next financial application.

Free and open source

Python is free in terms of license. Documentation is widely available, and many Python online community groups are available, where one can turn in times of doubt. Because it is free and open source, anyone can easily view or modify the algorithms in order to adapt to customized solutions.

Being accessible to the public opens a whole new level of opportunities. Anyone can contribute existing enhancements or create new modules. For advanced users, interoperability between different programming languages is supported. A Python interpreter may be embedded in C and C++ programs. Likewise, with the appropriate libraries, Python may be integrated with other languages not limited to Fortran, Lisp, PHP, Lua, and more.

Python is available on all major operating systems, such as Windows, Unix, OS/2, Mac, among others.

High-level, powerful, and flexible

Python as a general-purpose, high-level programming language allows the user to focus on problem solving and leave low-level mechanical constructs such as memory management out of the picture.

The expressiveness of the Python programming language syntax helps quantitative developers in implementing prototypes quickly.

Python allows the use of object-oriented, procedural, as well as functional programming styles. Because of this flexibility, it is especially useful in implementing complex mathematical models containing multiple changeable parameters.

A wealth of standard libraries

By now, you should be familiar with the `NumPy`, `SciPy`, `matplotlib`, `statsmodels`, and `pandas` modules, as indispensable tools in quantitative analysis and data management.

Other libraries extend the functionalities of Python. For example, one may turn Python into a data visualization tool with the **gnuplot** package in visualizing mathematical functions and data interactively. With Tk-based GUI tools such as **Tkinter**, it is possible to turn Python scripts into GUI programs.

A widely popular shell for Python is IPython, which provides interactive computing and high-performance tools for parallel and distributed computing. With IPython Notebook, the rich text web interface of IPython, you can share code, text, mathematical expressions, plots, and other rich media with your target audience. IPython was originally intended for scientists to work with Python and data.

Objected-oriented versus functional programming

If you are working as a programmer in the finance industry, chances are that your program will be built for handling thousands or millions of dollars' worth in transactions. It is crucial that your programs are absolutely free of errors. More often than not, bugs arise due to unforeseen circumstances. As financial software systems and models become larger and more complex, practicing good software design is crucial. While writing the Python code, you may want to consider the object-oriented approach or the functional approach to structure your code for better readability.

The object-oriented approach

As the demand for clarity, speed, and flexibility in your program increases, it is important to keep your code readable, manageable, and lean. One popular technical approach to building software systems is by applying the object-oriented paradigm. Consider the following example of displaying a greeting message as a class:

```
class Greeting(object):

    def __init__(self, my_greeting):
        self.my_greeting = my_greeting

    def say_hello(self, name):
        print "%s %s" % (self.my_greeting, name)
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/` support and register to have the files e-mailed directly to you.

We created a class called `Greeting` that is capable of accepting an input argument in its constructor. For this example, we will define our greeting as `"Hello"`. The `say_hello` function is invoked with an input name and prints our greeting messages as follows:

```
>>> greeting = Greeting("Hello")
>>> greeting.say_hello("World")
>>> greeting.say_hello("Dog")
>>> greeting.say_hello("Cat")
Hello World
Hello Dog
Hello Cat
```

The functional approach

We can achieve the same `Greeting` functionality using the functional approach. Functional programming is a programming paradigm, where computer programs are structured and styled in such a way that they can be evaluated as mathematical functions. These pseudo mathematical functions avoid changing state data, while increasing reusability and brevity.

In Python, a function object can be assigned to a variable and, like any other variables, can be passed into functions as an argument as well as return its value.

Let's take a look at the following code that gives us the same output:

```
from functools import partial
def greeting(my_greeting, name):
    print "%s %s" % (my_greeting, name)
```

Here, we defined a function named `greeting` that takes in two arguments. Using the `partial` function of the `functools` module, we treated the function, `greeting`, as an input variable, along with our variable greeting message as `Hello`.

```
>>> say_hello_to = partial(greeting, "Hello")
>>> say_hello_to("World")
>>> say_hello_to("Dog")
>>> say_hello_to("Cat")
```

We assigned the `say_hello_to` variable as its return value, and reused this variable in printing our greetings with three different names by executing it as a function that accepts input arguments.

Which approach should I use?

There is no clear answer to this question. We have just demonstrated that Python supports both the object-oriented approach and the functional approach. We can see that in certain circumstances the functional approach in software development is brevity to a large extent. Using the `say_hello_to` function provides better readability over `greeting.say_hello()`. It boils down to the programmer's decision as to what works best in making the code more readable and having ease of maintenance during the software life cycle while collaborating with fellow developers.

As a general rule of thumb, in large and complex software systems representing objects as classes helps in code management between team members. By working with classes, the scope of work can be more easily defined, and system requirements can be easily scaled using object-oriented design. When working with financial mathematical models, using functional programing helps to keep the code working in the same fashion as its accompanying mathematical concepts.

Which Python version should I use?

The code examples in this book have been tested in Python 2.7 but are optimized to run on Python 3. Many of the third-party Python modules mentioned in this book require at least Python 2.7, and some do not have support for Python 3 as yet. To achieve the best compatibility, it is recommended that you install Python 2.7 on your workstation.

If you have not installed Python on your workstation, you can find out more about Python from the official source at `https://www.python.org`. However, in order to build financial applications from the examples, you are required to use a number of additional third-party Python modules such as `NumPy`, `SciPy`, and `pandas`. It is recommended that you obtain an all-in-one installer to ease the installation procedures. The following are some popular installers that include hundreds of supported packages:

- Anaconda by Continuum Analytics at `https://store.continuum.io/cshop/anaconda`
- Canopy by Enthought at `https://store.enthought.com`

Introducing IPython

IPython is an interactive shell with high-performance tools used for parallel and distributed computing. With the IPython Notebook, you can share code, text, mathematical expressions, plots, and other rich media with your target audience.

In this section, we will learn how to get started and run a simple IPython Notebook.

Getting IPython

Depending on how you have installed Python on your machine, IPython might have been included in your Python environment. Please consult the IPython official documentation for the various installation methods most comfortable for you. The official page is available at `http://ipython.org`.

IPython can be downloaded from `https://github.com/ipython`. To install IPython, unpack the packages to a folder. From the terminal, navigate to the top-level source directory and run the following command:

```
$ python setup.py install
```

Using pip

The `pip` tool is a great way to install Python packages automatically. Think of it as a package manager for Python. For example, to install IPython without having to download all the source files, just run the following command in the terminal:

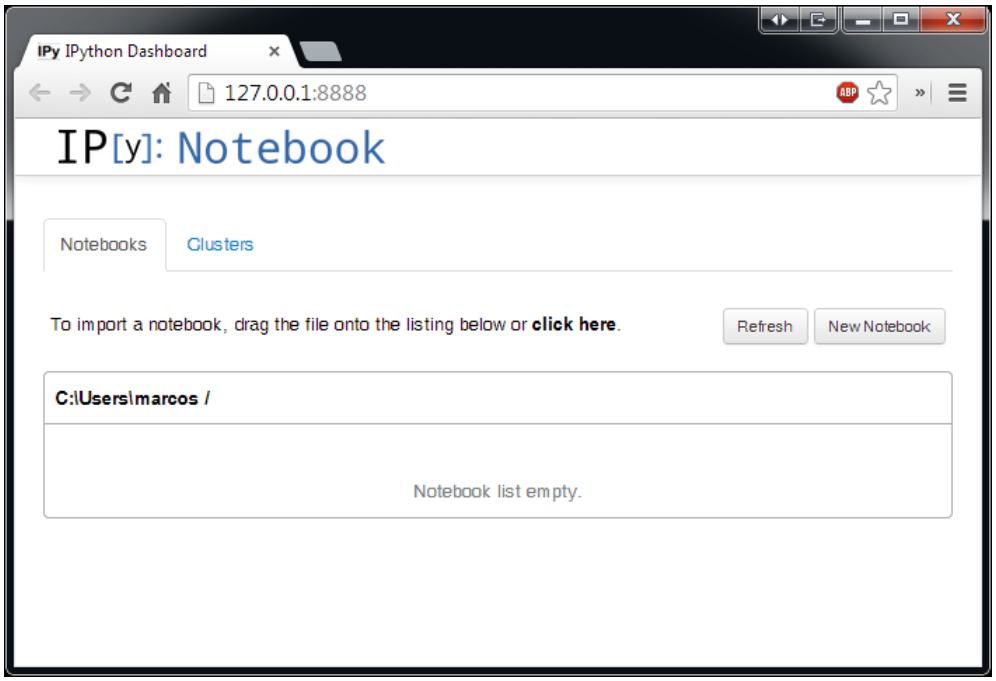
```
$ pip install ipython
```

To get pip to work in the terminal, it has to be installed as a Python module. Instructions for downloading and installing pip can be found at `https://pypi.python.org/pypi/pip`.

The IPython Notebook

The IPython Notebook is the web-based interactive computing interface of IPython used for the whole computation process of developing, documenting, and executing the code. This section covers some of the common features in IPython Notebook that you may consider using for building financial applications.

Here is a screenshot of the IPython Notebook in Windows OS:



Notebooks allow in-browser editing and executing of the code with the outputs attached to the code that generated them. It has the capability of displaying rich media, including images, videos, and HTML components.

Its in-browser editor allows the Markdown language that can provide rich text and commentary for the code.

Mathematical notations can be included with the use of LaTeX, rendered natively by MathJax. With the ability to import Python modules, publication-quality figures can be included inline and rendered using the `matplotlib` library.

Notebook documents

Notebook documents are saved with the `.ipynb` extension. Each document contains everything related to an interactive session, stored internally in the JSON format. Since JSON files are represented in plain text, this allows notebooks to be version controlled and easily sharable.

Notebooks can be exported to a range of static formats, including HTML, LaTeX, PDF, and slideshows.

Notebooks can also be available as static web pages and served to the public via URLs using `nbviewer` (IPython Notebook Viewer) without users having to install Python. Conversion is handled using the `nbconvert` service.

Running the IPython Notebook

Once you have IPython successfully installed in the terminal, type the following command:

```
$ ipython notebook
```

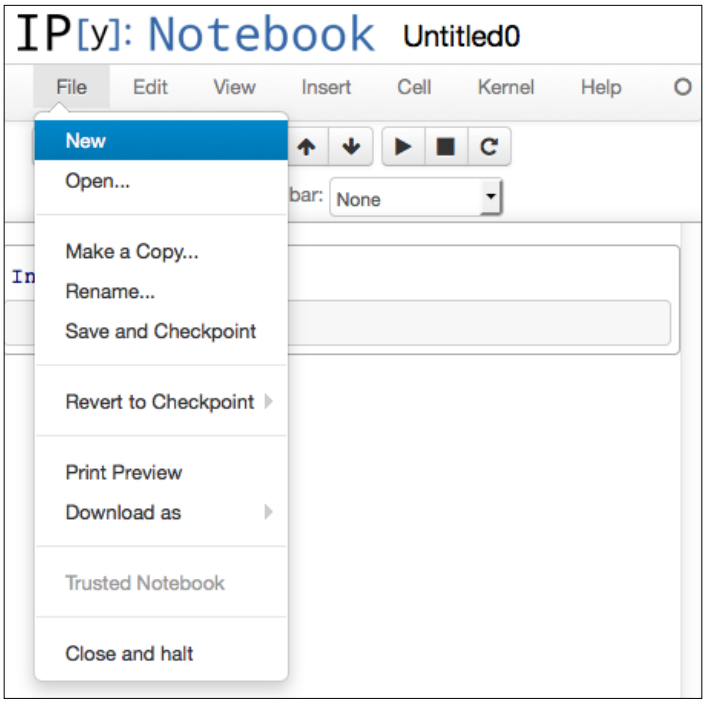
This will start the IPython Notebook server service , which runs in the terminal. By default, the service will automatically open your default web browser and navigate to the landing page. To access the notebook program manually, enter the `http://localhost:8888` URL address in your web browser.

By default, a notebook runs in port 8888. To infer the correct notebook address, check the log output from the terminal.

The landing page of the notebook web application is called the **dashboard**, which shows all notebooks currently available in the notebook directory. By default, this directory is the one from which the notebook server was started.

Creating a new notebook

Click on **New Notebook** from the dashboard to create a new notebook. You can navigate to the **File** | **New** menu option from within an active notebook:



Here, you will be presented with the notebook name, a menu bar, a toolbar, and an empty code cell.

The menu bar presents different options that may be used to manipulate the way the notebook functions.

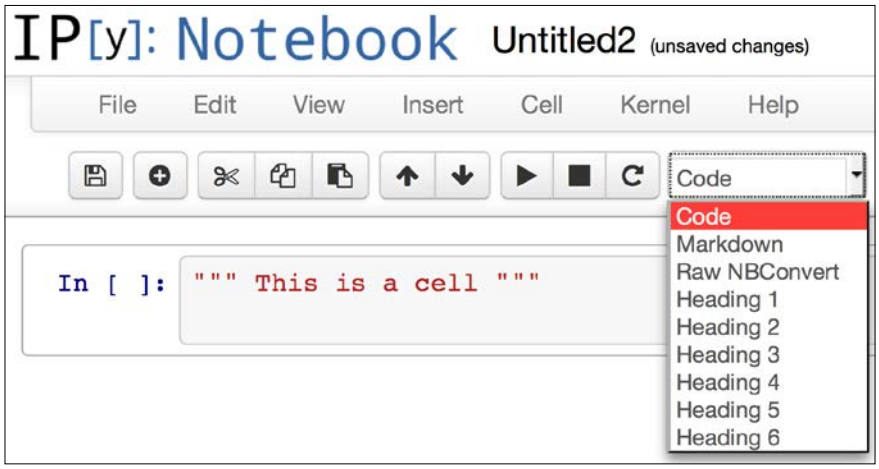
The toolbar provides shortcuts to frequently used notebook operations in the form of icons.

Notebook cells

Each logical section in a notebook is known as a **cell**. A cell is a multi-line text input field that accepts plain text. A single notebook document contains at least one cell and can have multiple cells.

To execute the contents of a cell, from the menu bar, go to **Cell | Run**, or click on the **Play** button from the toolbar, or use the keyboard shortcut *Shift + Enter*.

Each cell can be formatted as a **Code**, **Markdown**, **Raw NBConvert**, or heading cell:



Code cell

By default, each cell starts off as a code cell, which executes the Python code when you click on the **Run** button. Cells with a rounded rectangular box and gray background accept text inputs. The outputs of the executed box are displayed in the white space immediately below the text input.

Markdown cell

Markdown cells accept the Markdown language that provides a simple way to format plain text into rich text. It allows arbitrary HTML code for formatting.

Mathematical notations can be displayed with standard LaTeX and AMS-LaTeX (the `amsmath` package). Surround the LaTeX equation with `$` to display inline mathematics, and `$$` to display equations in a separate block. When executed, MathJax will render Latex equations with high-quality typography.

Raw NBConvert cell

Raw cells provide the ability to write the output directly and are not evaluated by the notebook.

Heading cells

Cells may be formatted as a heading cell, from level 1 (top level) to level 6 (paragraph). These are useful for the conceptual structure of your document or to construct a table of contents.

Simple exercises with IPython Notebook

Let's get started by creating a new notebook and populating it with some content. We will insert the various types of objects to demonstrate the various tasks.

Creating a notebook with heading and Markdown cells

We will begin by creating a new notebook by performing the following steps:

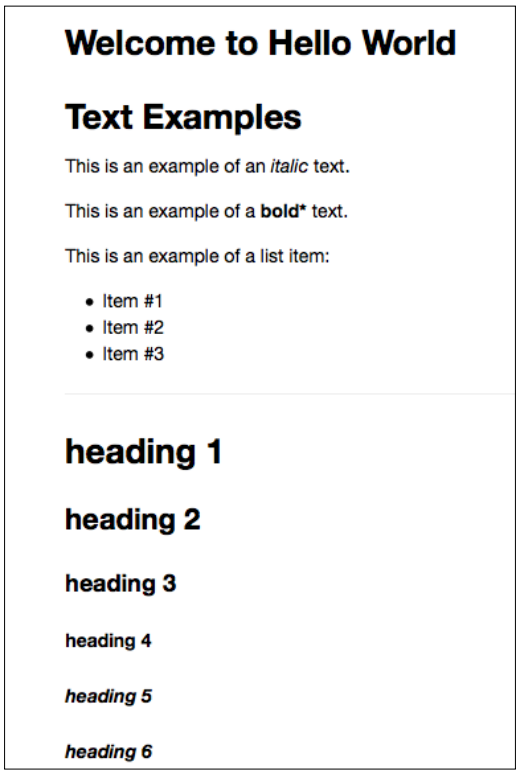
1. Click on **New Notebook** from the dashboard to create a new notebook. If from within an active notebook, navigate to the **File** | **New** menu option.
2. In the input field of the first cell, enter a page title for this notebook. In this example, type in `Welcome to Hello World`.
3. From the options toolbar menu, go to **Cells** | **Cell Type** and select **Heading 1**. This will format the text we have entered as the page title. The changes, however, will not be immediate at this time.
4. From the options toolbar menu, go to **Insert** | **Insert Cell Below**. This will create another input cell below our current cell.
5. In this example, we will insert the following piece of text that contains the Markdown code:

```
Text Examples
===

This is an example of an *italic* text.

This is an example of a **bold*** text.

This is an example of a list item:
- Item #1
- Item #2
- Item #3

---

#heading 1
##heading 2
###heading 3
####heading 4
#####heading 5
######heading 6
```

6. From the toolbar, select **Markdown** instead of **Code**.
7. To run your code, go to **Cell** | **Run All**. This option will run all the Python commands and format your cells as required.

When the current cell is executed successfully, the notebook will focus on the next cell below, ready for your next input. If no cell is available, one will be automatically created and will receive the input focus.

This will give us the following output:



Welcome to Hello World

Text Examples

This is an example of an *italic* text.

This is an example of a **bold*** text.

This is an example of a list item:

- Item #1
- Item #2
- Item #3

heading 1

heading 2

heading 3

heading 4

heading 5

heading 6

Saving notebooks

Go to **File** and click on **Save** and **Checkpoint**. Our notebook will be saved as an `.ipynb` file.

Mathematical operations in cells

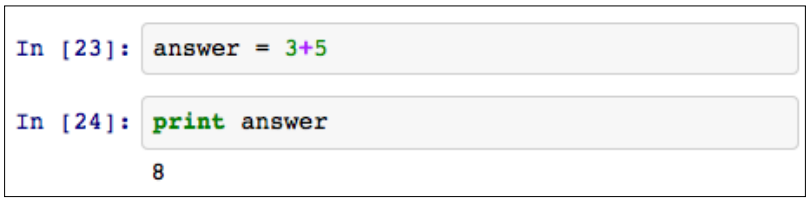
Let's perform a simple mathematical calculation in the notebook; let's add the numbers 3 and 5 and assign the result to the `answer` variable by typing in the code cell:

```
answer = 3 + 5
```

From the options menu, go to **Insert** | **Insert Cell Below** to add a new code cell at the bottom. We want to output the result by typing in the following code in the next cell:

```
print answer
```

Next, go to **Cell** | **Run All**. Our answer is printed right below the current cell.



Displaying graphs

The `matplotlib` module provides a MATLAB-like plotting framework in Python. With the `matplotlib.pyplot` function, charts can be plotted and rendered as graphic images for display in a web browser.

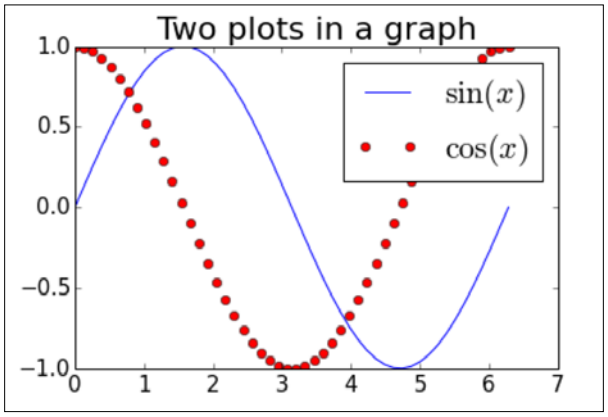
Let's demonstrate a simple plotting functionality of the IPython Notebook. In a new cell, paste the following code:

```
import numpy as np
import math
import matplotlib.pyplot as plt

x = np.linspace(0, 2*math.pi)
plt.plot(x, np.sin(x), label=r'$\sin(x)$')
plt.plot(x, np.cos(x), 'ro', label=r'$\cos(x)$')
plt.title(r'Two plots in a graph')
plt.legend()
```

The first three lines of the code contain the required import statements. Note that the `NumPy`, `math`, and `matplotlib` packages are required for the code to work in the IPython Notebook.

In the next statement, the variable x represents our x axis values from 0 through 7 in real numbers. The following statement plots the `sine` function for every value of x. The next `plot` command plots the `cosine` function for every value of x as a dotted line. The last two lines of the code print the title and legend respectively.

Running this cell gives us the following output:



Inserting equations

What is TeX and LaTeX? **TeX** is the industry standard document markup language for math markup commands. **LaTeX** is a variant of TeX that separates the document structure from the content.

Mathematical equations can be displayed using LaTeX in the Markdown parser. The IPython Notebook uses MathJax to render LaTeX surrounded with `$$` inside Markdown.

For this example, we will display a standard normal cumulative distribution function by typing in the following command in the cell:

```
$$N(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-
\frac{z^2}{2}}\, dz$$
```

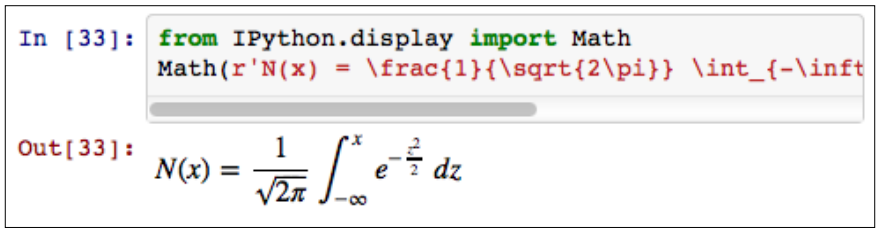
Select **Markdown** from the toolbar and run the current cell. This will transform the current cell into its respective equation output:

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{z^2}{2}}\, dz$$

Besides using the MathJax typesetting, another way of displaying the same equation is using the `math` function of the IPython display module, as follows:

```
from IPython.display import Math
Math(r'N(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-
\frac{z^2}{2}}\, dz')
```

The preceding code will display the same equation, as shown in the following screenshot:

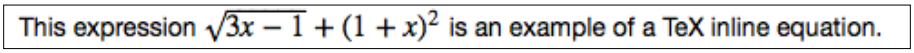


Notice that, since this cell is run as a normal code cell, the output equation is displayed immediately below the code cell.

We can also display equations inline with text. For example, we will use the following code with a single `$` wrapping around the LaTeX expression:

```
This expression $\sqrt{3x-1}+(1+x)^2$ is an example of a TeX
inline equation
```

Run this cell as the **Markdown** cell. This will transform the current cell into the following:



Displaying images

To work with images, such as JPEG and PNG, use the `Image` class of the IPython display module. Run the following code to display a sample image:

```
from IPython.display import Image
Image(url='http://python.org/images/python-logo.gif')
```

On running the code cell, it will display the following output:

```
In [15]: from IPython.display import Image
         Image(url='http://python.org/images/python-logo.gif')
Out[15]:
```

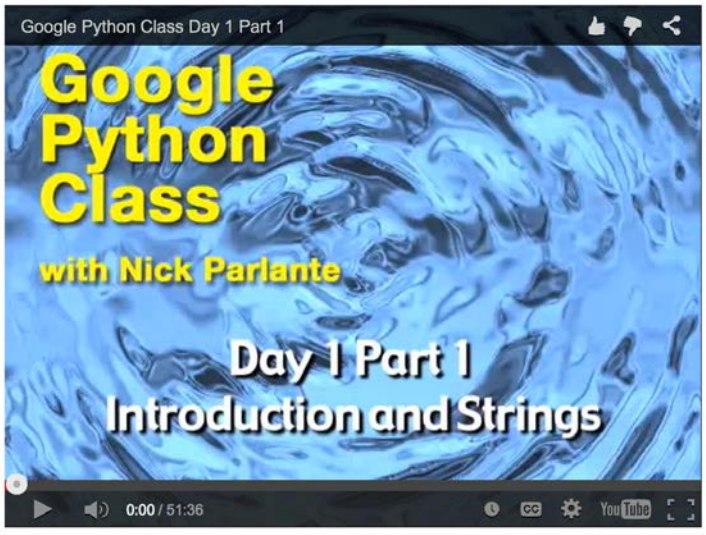
Inserting YouTube videos

The `lib.display` module of the `IPython` package contains a `YouTubeVideo` function, where you can embed videos hosted externally on YouTube into your notebook. For example, run the following code:

```
from IPython.lib.display import YouTubeVideo

# An introduction to Python by Google.
YouTubeVideo('tKTZoB2Vjuk')
```

The video will be displayed below the code, as shown in the following screenshot:

```
In [48]: from IPython.lib.display import YouTubeVideo

         # An introduction to Python by Google.
         YouTubeVideo('tKTZoB2Vjuk')
Out[48]:
```

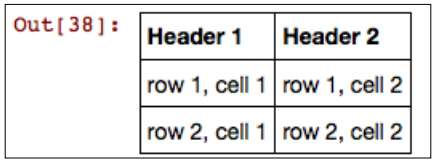


Working with HTML

Notebook allows HTML representations to be displayed. One common use of HTML is the ability to display data with tables. The following code outputs a table with two columns and three rows, including a header row:

```
from IPython.display import HTML
table = """<table>
<tr>
<th>Header 1</th>
<th>Header 2</th>
</tr>
<tr>
<td>row 1, cell 1</td>
<td>row 1, cell 2</td>
</tr>
<tr>
<td>row 2, cell 1</td>
<td>row 2, cell 2</td>
</tr>
</table>"""
HTML(table)
```

The `HTML` function will render HTML tags as a string in its input argument. We can see the final output as follows:



The pandas DataFrame object as an HTML table

In a notebook, pandas allow DataFrame objects to be represented as HTML tables.

In this example, we will retrieve the stock market data from Yahoo! Finance and store them in a pandas DataFrame object with the help of the `panas.io.data.web.DataReader` function. Let's use the `AAPL` ticker symbol as the first argument, `yahoo` as its second argument, the start date of the market data as the third argument, and the end date as the last argument:

```
import pandas.io.data as web
import datetime
```

```
start = datetime.datetime(2014, 1, 1)
end = datetime.datetime(2014, 12, 31)
df = web.DataReader("AAPL", 'yahoo', start, end)
df.head()
```

With the `df.head()` command, the first five rows of the DataFrame object that contains the market data is displayed as an HTML table in the notebook:

```
In [78]: import pandas.io.data as web
         import datetime

In [79]: start = datetime.datetime(2014, 1, 1)
         end = datetime.datetime(2014, 12, 31)

In [80]: df = web.DataReader("AAPL", 'yahoo', start, end)
         df.head()
```

Out[80]:

	Open	High	Low	Close	Volume	Adj Close
Date						
2014-01-02	555.68	557.03	552.02	553.13	58671200	77.39
2014-01-03	552.86	553.70	540.43	540.98	98116900	75.69
2014-01-06	537.45	546.80	533.60	543.93	103152700	76.10
2014-01-07	544.32	545.96	537.92	540.04	79302300	75.56
2014-01-08	538.81	545.56	538.69	543.46	64632400	76.04

5 rows × 6 columns

Notebook for finance

You are now ready to place your code in a chronological order and present the key financial information, such as plots and data to your audience. Many industry practitioners use the IPython Notebook as their preferred editor for financial model development in helping them to visualize data better.

You are strongly encouraged to explore the powerful features the IPython Notebook has to offer that best suit your modeling needs. A gallery of interesting notebook projects used in scientific computing can be found at `https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks`.

Summary

In this chapter, we discussed how Python might be suitable for certain areas of finance and also discussed its advantages for our software applications. We also considered the functional programming paradigm and the object-oriented programming paradigm that are supported in Python, and saw how we can achieve brevity in our applications. There is no clear rule as to how one approach may be favored over the other. Ultimately, Python gives programmers the flexibility to structure their code to the best interests of the project at hand.

We were introduced to IPython, the interactive computing shell for Python, and explored its usefulness in scientific computing and rich media presentation. We worked on simple exercises on our web browser with the IPython Notebook, and learned how to create a new notebook document, insert text with the Markdown language, performed simple calculations, plotted graphs, displayed mathematical equations, inserted images and videos, rendered HTML, and learned how to use pandas to fetch the stock market data from Yahoo! Finance as a DataFrame object before presenting its content as an HTML table. This will help us visualize data and perform rich media presentations to our audience.

Python is just one of the many powerful programing languages that can be considered in quantitative finance studies, not limited to Julia, R, MATLAB, and Java. You should be able to present key concepts more effectively in the Python language. These concepts, once mastered, can easily be applied to any language you choose when creating your next financial application.

In the next chapter, we will explore linear models in finance and techniques used in portfolio management.

2
The Importance of Linearity in Finance

Nonlinear dynamics plays a vital role in our world. Linear models are often employed in economics due to their simplicity for studies and easier modeling capabilities. In finance, linear models are widely used to help price securities and perform optimal portfolio allocation, among other useful things. One of the significance of linearity in financial modeling is its assurance that a problem terminates at a global optimal solution.

In order to perform prediction and forecasting, regression analysis is widely used in the field of statistics to estimate relationships among variables. With an extensive mathematics library being one of Python's greatest strength, Python is frequently used as a scientific scripting language to aid in these problems. Modules such as the `SciPy` and `NumPy` packages contain a variety of linear regression functions for data scientists to work with.

In traditional portfolio management, the allocation of assets follows a linear pattern, and investors have individual styles of investing. We can state the problem of portfolio allocation into a system of linear equations, containing either equalities or inequalities. These linear systems can then be represented in a matrix form as $Ax = B$, where A is our known coefficient value, B is the observed result, and x is the vector of values that we want to find out. More often than not, x contains the optimal security weights to maximize our utility. Using matrix algebra, we can efficiently solve for x using either direct or indirect methods.

In this chapter, we will cover the following topics:

- Examining the CAPM model with the efficient frontier and capital market line
- Solving for the security market line using regression

- Examining the APT model and performing a multivariate linear regression
- Understanding linear optimization in portfolio allocation
- Linear optimization using the PuLP package
- Understanding the outcomes of linear programming
- Introduction to integer programming
- Implementing a linear integer programming model with binary conditions
- Solving systems of linear equations with equalities using matrix linear algebra
- Solving systems of linear equations directly with the LU, Cholesky, and QR decomposition
- Solving systems of linear equations indirectly with the Jacobi and Gauss-Seidel method

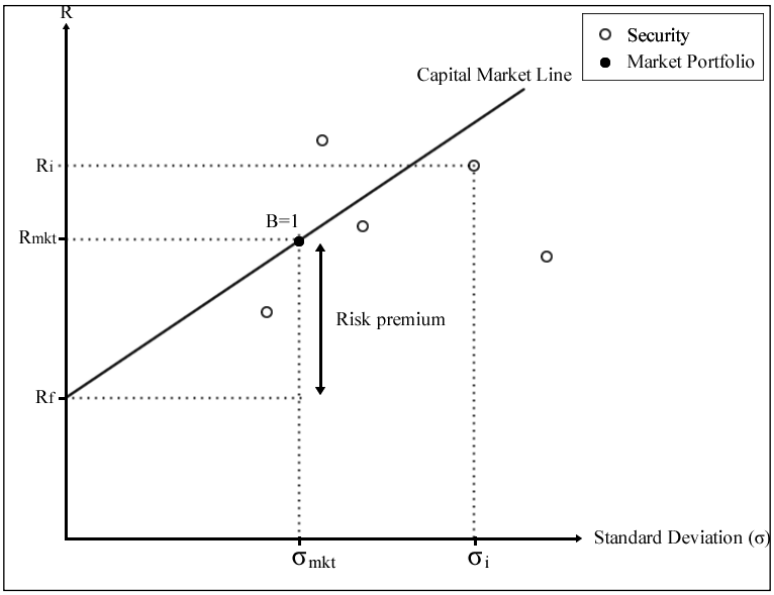
The capital asset pricing model and the security market line

Many financial literatures devote exclusive discussions to the **capital asset pricing model** (**CAPM**). In this section, we will explore the key concepts that highlight the importance of linearity in finance.

In the famous CAPM, the relationship between risk and rates of returns in a security is described as follows:

$$R_i = R_f + \beta_i \left(R_{mkt} - R_f \right)$$

For a security i, its returns is defined as R_i and its beta as β_i. The CAPM defines the return of the security as the sum of the risk-free rate R_f and the multiplication of its beta with the risk premium. The risk premium can be thought of as the market portfolio's excess returns exclusive of the risk-free rate. The following figure is a visual representation of the CAPM:



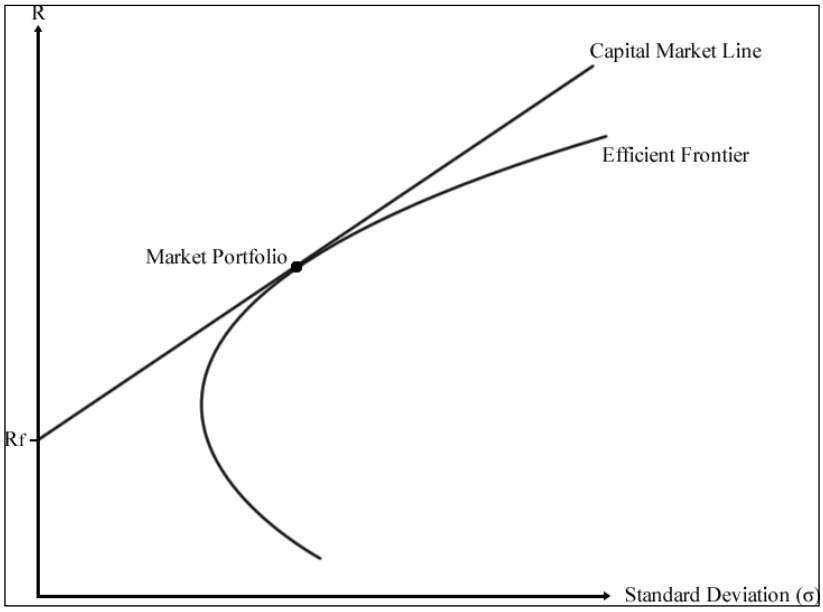
Beta is a measure of the systematic risk of a stock; a risk that cannot be diversified away. In essence, it describes the sensitivity of stock returns with respect to movements in the market. For example, a stock with a beta of zero produces no excess returns regardless of the direction the market moves—it can only grow at the risk-free rate. A stock with a beta of 1 indicates that the stock moves perfectly with the market.

The beta is mathematically derived by dividing the covariance of returns between the stock and the market with the variance of the market returns.

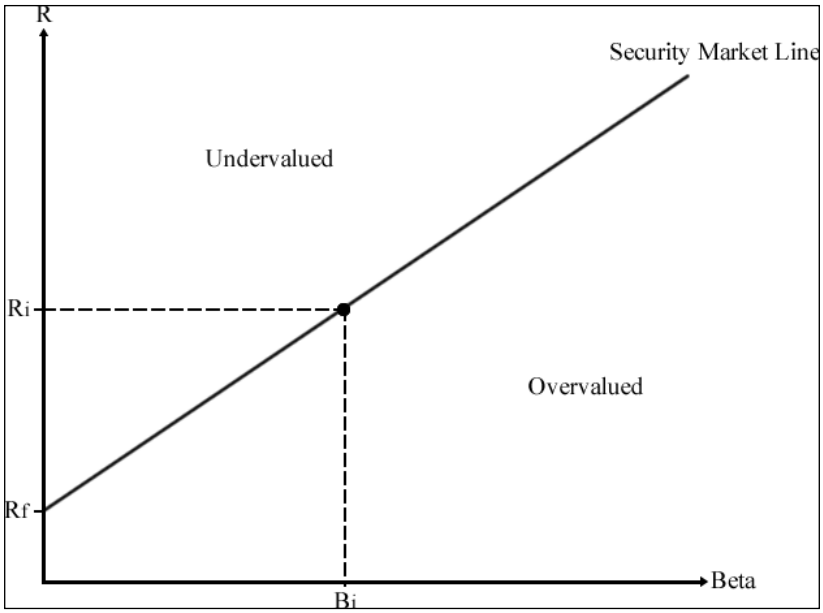
The CAPM model measures the relationship between risk and stock returns for every stock in the portfolio basket. By outlining the sum of this relationship, we obtain combinations or weights of risky securities that produce the lowest portfolio risk for every level of portfolio return. An investor who wishes to receive a particular return would own one such combination of an optimal portfolio that provides the least risk possible. The combinations of optimal portfolios lie along a line called the **efficient frontier**.

Along the efficient frontier, there exists a tangent point that denotes the best optimal portfolio available giving the highest rate of return in exchange for the lowest risk possible. This optimal portfolio at the tangent point is known as the **market portfolio**.

There exists a straight line drawn from the market portfolio to the risk-free rate. This line is called the **capital market line** (**CML**). The CML can be thought of as the highest Sharpe ratio available among all the other Sharpe ratios of optimal portfolios. The Sharpe ratio is a risk-adjusted performance measure defined as the portfolio's excess returns over the risk-free rate per unit of its risk in standard deviations. Investors are particularly interested in holding the combinations of assets along the CML line. Let's take a look at the following graphical figure:



Another line of interest in CAPM studies is the **security market line** (**SML**). The SML plots the asset's expected returns against its beta. For a security with a beta value of 1, its returns perfectly match the market's returns. Any security priced above the SML is deemed to be undervalued since investors expect a higher return given the same amount of risk. Conversely, any security priced below the SML is deemed to be overvalued:



Suppose we are interested in finding the beta β_i of a security. We can regress the company's stock returns r_i against the market's returns r_M along with an intercept α in the form the equation $r_i = \alpha + \beta r_M$.

Consider the set of stock return and market return data measured over five time periods:

Time period	Stock returns	Market returns
1	0.065	0.055
2	0.0265	-0.09
3	-0.0593	-0.041
4	-0.001	0.045
5	0.0346	0.022

Using the `stats` module of `SciPy`, we will perform a least squares regression on the CAPM model, and derive the values of α and β_i by running the following code in Python:

```
""" Linear regression with SciPy """
from scipy import stats

stock_returns = [0.065, 0.0265, -0.0593, -0.001, 0.0346]
```

```
mkt_returns = [0.055, -0.09, -0.041, 0.045, 0.022]

beta, alpha, r_value, p_value, std_err = \
       stats.linregress(stock_returns, mkt_returns)
```

The `scipy.stats.linregress` function returns the slope of the regression line, the intercept of the regression line, the correlation coefficient, the p-value for a hypothesis test with null hypothesis of a zero slope, and the standard error of the estimate. We are interested in finding the slope and intercept of the line:

```
>>> print beta, alpha
0.507743187877  -0.00848190035246
```

The beta of the stock is 0.5077.

The equation describing the SML can be written as:

$$E\left(R_i\right) = R_f + \beta_i\left[E\left(R_M\right) - R_f\right]$$

The term $E\left(R_M\right) - R_f$ is the market risk premium, and $E\left(R_M\right)$ is the expected return on the market portfolio. R_f is the return on the risk-free rate, $E\left(R_i\right)$ is the expected return on asset i, and β_i is the beta of asset.

Suppose the risk-free rate is 5 percent and the market risk premium is 8.5 percent. What is the expected return of the stock? Based on the CAPM, the equity with a beta of 0.5077 would have a risk premium of $0.5077 \times 8.5\%$, or 4.3 percent. The risk-free rate is 5 percent, so the expected return on the equity is 9.3 percent.

Should the security be observed in the same time period to have a higher return (for example, 10.5 percent) than the expected stock return, the security can be said to be undervalued, since the investor can expect greater return for the same amount of risk.

Conversely, should the return of the security be observed to have a lower return (for example, 7 percent) than the expected return as implied by the SML, the security can be said to be overvalued. The investor receives less return for assuming the same amount of risk.

The Arbitrage Pricing Theory model

The CAPM suffers from several limitations, such as the use of a mean-variance framework and the fact that returns are captured by one risk factor—the market risk factor. In a well-diversified portfolio, the unsystematic risk of various stocks cancels out and is essentially eliminated.

The **Arbitrage Pricing Theory** (**APT**) model was put forward to address these shortcomings and offers a general approach of determining the asset prices other than the mean and variances.

The APT model assumes that the security returns are generated according to multiple factor models, which consist of a linear combination of several systematic risk factors. Such factors could be the inflation rate, GDP growth rate, real interest rates, or dividends.

The equilibrium asset pricing equation according to the APT model is as follows:

$$E\left[R_i\right] = \alpha_i + \beta_{i,1}F_1 + \beta_{i,2}F_2 + \cdots + \beta_{i,j}F_j$$

Here, $E\left[R_i\right]$ is the expected rate of return on security i, α_i is the expected return on stock $i$if all factors are negligible, $\beta_{i,j}$ is the sensitivity of the ith asset to the jth factor, and F_j is the value of the jth factor influencing the return on stock i.

Since our goal is to find all values of α_i and β, we will perform a **multivariate linear regression** on the APT model.

Multivariate linear regression of factor models

Many Python packages such as `SciPy` come with several variants of regression functions. In particular, the `statsmodels` package is a complement to `SciPy` with descriptive statistics and estimation of statistical models. The official page for `statsmodels` is `http://statsmodels.sourceforge.net/`.

In this example, we will use the `ols` function of the `statsmodels` module to perform an ordinary least squares regression and view its summary.

Let's assume that you have implemented an APT model with seven factors that return the values of *Y*. Consider the following set of data collected over 9 time periods, *t*1 to *t*9. *X1* to *X7* are independent variables observed at each period. The regression problem is therefore structured as: $Y = X_{l,1}F_1 + X_{l,2}F_2 + \cdots + X_{l,7}F_7 + c$.

A simple ordinary least squares regression on values of *X* and *Y* can be performed with the following code:

```
""" Least squares regression with statsmodels """
import numpy as np
import statsmodels.api as sm

# Generate some sample data
num_periods = 9
all_values = np.array([np.random.random(8)
                       for i in range(num_periods)])

# Filter the data
y_values = all_values[:, 0]  # First column values as Y
x_values = all_values[:, 1:]  # All other values as X

x_values = sm.add_constant(x_values)  # Include the intercept
results = sm.OLS(y_values, x_values).fit()  # Regress and
fit the model
```

Let's view the detailed statistics of the regression:

```
>>> print results.summary()
```

The OLS regression results will output a pretty long table of statistical information. However, our interest lies in the particular section that gives us the coefficients of our APT model:

```
==============================================
                 coef    std err          t
----------------------------------------------
const          0.5224      0.825      0.633
x1             0.0683      0.246      0.277
x2             0.1455      1.010      0.144
x3            -0.2451      0.330     -0.744
x4             0.5907      0.830      0.712
x5            -0.3252      0.256     -1.271
x6            -0.2375      0.788     -0.301
x7            -0.1880      0.703     -0.267
```

Similarly, we can use the `params` function to display our coefficients of interest:

```
>>> print results.params
[ 0.52243605  0.06827488  0.14550665 -0.24508947  0.5907154
-0.32515442  -0.23751989  -0.18795065]
```

Both the function calls produce the same coefficient values for the APT model in the same order.

Linear optimization

In the CAPM and APT pricing theories, we assumed linearity in the models and solved for expected security prices using regressions in Python.

As the number of securities in our portfolio increases, certain limitations are introduced as well. A portfolio manager would find himself constrained by these rules in pursing certain objectives mandated by investors.

Linear optimization helps you overcome the problem of portfolio allocation. Optimization focuses on minimizing or maximizing the value of the objective functions. The examples are maximizing returns and minimizing volatility. These objectives are usually governed by certain regulations, such as no short-selling rule, limits on the number of securities to be invested, and so on.

Unfortunately, in Python there is no single official package that supports this solution. However, there are third-party packages available with the implementation of the simplex algorithm for linear programming. For the purpose of this demonstration, we will use PuLP, an open source linear programming modeler, to assist us in this particular linear programming problem.

Getting PuLP

You can obtain PuLP from `https://github.com/coin-or/pulp`. The project page contains a comprehensive list of documentations to help you get started with your optimization process.

A simple linear optimization problem

Suppose that we are interested in investing in two securities, x and y. We would like to find out the actual number of units to invest for every 3 units of security *X* and 2 units of security *Y*, such that the total number of units invested is maximized, where possible. However, there are certain constraints on our investment strategy:

- For every 2 units of security *X* invested and 1 unit of security *Y* invested, the total volume must not exceed 100
- For every unit of security *X* and *Y* invested, the total volume must not exceed 80
- The total volume allowed to invest in security *X* must not exceed 40
- Short selling is not allowed for both the securities

The maximization problem can be mathematically represented as follows:

$$\text{Maximize: } f(x,y) = 3x+2y$$

$$\textit{Subject to}:$$

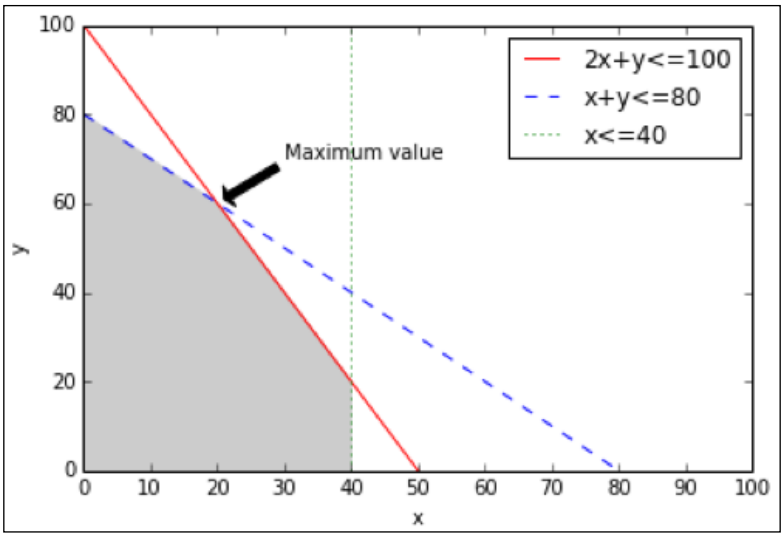
$$2x + y \leq 100$$

$$x + y \leq 80$$

$$x \leq 40$$

$$x \geq 0, y \geq 0$$

By plotting the constraints on an x by y graph, the set of feasible solutions is shown in the shaded area:



The problem can be translated to Python with the PuLP package:

```
""" A simple linear optimization problem with 2 variables """
import pulp

x = pulp.LpVariable("x", lowBound=0)
y = pulp.LpVariable("y", lowBound=0)

problem = pulp.LpProblem("A simple maximization objective",
                         pulp.LpMaximize)
problem += 3*x + 2*y, "The objective function"
problem += 2*x + y <= 100, "1st constraint"
problem += x + y <= 80, "2nd constraint"
problem += x <= 40, "3rd constraint"
problem.solve()
```

The `LpVariable` function defines a variable to be solved. The `LpProblem` function initializes the problem with a text description of the problem and the type of optimization, which in this case is the maximization method. The `+=` operation allows an arbitrary number of constraints to be added, along with a text description. Finally, the `solve` function is called to begin performing linear optimization. Each variable value is printed to show the values that the optimizer has solved for us.

The following output is generated when the code runs:

```
>>> print "Maximization Results:"
>>> for variable in problem.variables():
...     print variable.name, "=", variable.varValue
Maximization Results:
x = 20.0
y = 60.0
```

The results show that obtaining the maximum value of 180 is possible when the value of x is 20 and y is 60 while fulfilling the given set of constraints.

Outcomes of linear programs

There are three outcomes in linear optimization, as follows:

1. A local optimal solution to a linear program is a feasible solution with a closer objective function value than all other feasible solutions close to it. It may or may not be the **global optimal solution**, which is a solution that is better than every feasible solution.
2. A linear program is **infeasible** if a solution cannot be found.
3. A linear program is **unbounded** if the optimal solution is unbounded or is infinite.

Integer programming

In the simple optimization problem we have investigated earlier, so far the variables have been allowed to be continuous or fractional. What if the use of fractional values or results is not realistic? This problem is called the **linear integer programming** problem, where all the variables are restricted as integers. A special case of an integer variable is a binary variable that can either be 0 or 1. Binary variables are especially useful to model decision making given a set of choices.

Integer programming models are frequently used in operational research to model real-world working problems. More often than not, stating nonlinear problems in a linear or even binary fashion requires more art than science.

An example of an integer programming model with binary conditions

Suppose we must go for 150 contracts in a particular over-the-counter exotic security from three dealers. Dealer *X* quoted $500 per contract plus handling fees of $4,000, regardless of the number of contracts sold. Dealer *Y* charges $450 per contract plus a transaction fee of $2,000. Dealer *Z* charges $450 per contract plus a fee of $6,000. Dealer *X* will sell at most 100 contracts, dealer *Y* at most 90, and dealer *Z* at most 70. The minimum transaction volume from any dealer is 30 contracts if any are transacted with that dealer. How should we minimize the cost of purchasing 150 contracts?

Using the `pulp` package, let's set up the required variables:

```
""" An example of implementing an integer programming model with
binary conditions """
import pulp

dealers = ["X", "Y", "Z"]
variable_costs = {"X": 500,
                  "Y": 350,
                  "Z": 450}
fixed_costs = {"X": 4000,
               "Y": 2000,
               "Z": 6000}

# Define PuLP variables to solve
quantities = pulp.LpVariable.dicts("quantity",
                                   dealers,
                                   lowBound=0,
                                   cat=pulp.LpInteger)
is_orders = pulp.LpVariable.dicts("orders",
                                  dealers,
                                  cat=pulp.LpBinary)
```

The `dealers` variable simply contains the dictionary identifiers that are used to reference lists and dictionaries later on. The `variable_costs` and `fixed_costs` variables are dictionaries that contain their respective contract cost and fees charged by each dealer. The PuLP solver solves for the values of `quantities` and `is_orders`, which are defined by the `LpVariable` function. The `dicts` function tells PuLP to treat the assigned variable as a dictionary object, using the `dealers` variable for referencing. Note that the `quantities` variable has a lower boundary of 0 that prevents us from entering a short position in any securities. The `is_orders` values are treated as binary objects, indicating whether we should enter into a transaction with any of the dealers.

What is the best approach to modeling this integer programming problem? At first glance, this seems fairly straightforward by applying this equation:

$$Minimize \sum_{i=x}^{i=z} IsOrder_i \left[variable\,cost_i \times quantity_i + fixed\,cost_i \right]$$

Where

$$IsOrder_i = \begin{cases} 1, if\ buying\ from\ dealer\ i \\ 0, if\ not\ buying\ from\ dealer\ i \end{cases}$$

$$30 \le quantity_x \le 100$$

$$30 \le quantity_y \le 90$$

$$30 \le quantity_z \le 70$$

$$\sum_{i=x}^{i=z} quantity_i = 150$$

The equation simply states that we want to minimize the total costs with the binary variable $IsOrder_i$, determining whether to account for the costs associated with buying from a specific dealer should we choose to.

Let's implement this model in Python:

```
"""
This is an example of implementing an integer programming model with
binary
variables
the wrong way.
"""
# Initialize the model with constraints
model = pulp.LpProblem("A cost minimization problem",
pulp.LpMinimize)
model += sum([(variable_costs[i] * quantities[i] +
               fixed_costs[i])*is_orders[i] for i in dealers]), \
         "Minimize portfolio cost"
model += sum([quantities[i] for i in dealers]) == 150, \
         "Total contracts required"
```

```
model += 30 <= quantities["X"] <= 100, "Boundary of total volume
of X"
model += 30 <= quantities["Y"] <= 90, "Boundary of total volume of
Y"
model += 30 <= quantities["Z"] <= 70, "Boundary of total volume of
Z"

model.solve()
```

What happens when we run the solver?

```
TypeError: Non-constant expressions cannot be multiplied
```

As it turned out, we were trying to perform multiplication on two unknown variables, `quantities` and `is_order`, which unknowingly led us to perform nonlinear programming. Such are the pitfalls encountered when performing integer programming.

A different approach with binary conditions

Another method of formulating the minimization objective is to place all unknown variables in a linear fashion such that they are additive:

$$Minimize \sum_{i=x}^{i=z} variable\ cost_i \times quantity_i + fixed\ cost_i \times IsOrder_i$$

Comparing with the previous objective equation, we would obtain the same fixed cost values. However, the unknown variable $quantity_i$ remains in the first term of the equation. Hence, the variable $quantity_i$ is required to be solved as a function of $IsOrder_i$ such that the constraints are stated as follows:

$$IsOrder_i \times 30 \geq quantity_x \leq IsOrder_i \times 100$$

$$IsOrder_i \times 30 \leq quantity_y \leq IsOrder_i \times 90$$

$$IsOrder_i \times 30 \leq quantity_z \leq IsOrder_i \times 70$$

Let's apply these formulas in Python:

```
"""
This is an example of implementing an IP model
with binary variables the correct way.
```

```
"""
# Initialize the model with constraints
model = pulp.LpProblem("A cost minimization problem",
                       pulp.LpMinimize)
model += sum([variable_costs[i]*quantities[i] +
              fixed_costs[i]*is_orders[i] for i in dealers]), \
         "Minimize portfolio cost"
model += sum([quantities[i] for i in dealers]) == 150, \
         "Total contracts required"
model += is_orders["X"]*30 <= quantities["X"] <= \
         is_orders["X"]*100, "Boundary of total volume of X"
model += is_orders["Y"]*30 <= quantities["Y"] <= \
         is_orders["Y"]*90, "Boundary of total volume of Y"
model += is_orders["Z"]*30 <= quantities["Z"] <= \
         is_orders["Z"]*70, "Boundary of total volume of Z"
model.solve()
```

What happens when we try to run the solver?

```
>>> print "Minimization Results:"
>>> for variable in model.variables():
...     print variable, "=", variable.varValue
>>>
>>> print "Total cost: %s" % pulp.value(model.objective)
Minimization Results:
orders_X = 0.0
orders_Y = 1.0
orders_Z = 1.0
quantity_X = 0.0
quantity_Y = 90.0
quantity_Z = 60.0
Total cost: 66500.0
```

The output tells us that buying 90 contracts from the dealer *Y* and 60 contracts from the dealer *Z* gives us the lowest possible cost of $66,500 while fulfilling all the other constraints.

As we can see, careful planning is required in the designing of integer programming models to arrive at an accurate solution in order for them to be useful in decision making.

Solving linear equations using matrices

In the previous section, we looked at solving a system of linear equations with inequality constraints. If a set of systematic linear equations has constraints that are deterministic, we can represent the problem as matrices and apply matrix algebra. Matrix methods represent multiple linear equations in a compact manner while using the existing matrix library functions.

Suppose we would like to build a portfolio consisting of 3 securities, *a*, *b* and *c*. The allocation of the portfolio must meet certain constraints: it must consist of 6 units of a long position in security *a*. With every 2 units of security *a*, 1 unit of security *b*, and 1 unit of security *c* invested, the net position must be 4 units long. With every 1 unit of security *a*, 3 units of security *b*, and 2 units of security *c* invested, the net position must be long 5 units.

To find out the number of securities to invest in, we can frame the problem mathematically as follows:

$$2a + b + c = 4$$

$$a + 3b + 2c = 5$$

$$a = 6$$

With all of the coefficients visible, the equations are as follows:

$$2a + 1b + 1c = 4$$

$$1a + 3b + 2c = 5$$

$$1a + 0b + 0c = 6$$

Taking the coefficients of the equations and representing them in a matrix form:

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 2 \\ 1 & 0 & 0 \end{bmatrix}, x = \begin{bmatrix} a \\ b \\ c \end{bmatrix}, B = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$$

The linear equations can now be stated as follows:

$$Ax = B$$

To solve for the vector x that contains the number of securities to invest in, the inverse of matrix A is taken and the equation is written as follows:

$$x = A^{-1}B$$

Using the `NumPy` arrays, the A and B matrices are assigned as follows:

```
""" Linear algebra with NumPy matrices """
import numpy as np

A = np.array([[2, 1, 1],
              [1, 3, 2],
              [1, 0, 0]])
B = np.array([4, 5, 6])
```

We can use the `linalg.solve` function of `NumPy` to solve a system of linear scalar equations:

```
>>> print np.linalg.solve(A, B )
[  6.  15. -23.]
```

The portfolio would require a long position of 6 units of security *a*, 15 units of security *b*, and a short position of 23 units of security *c*.

In portfolio management, we can use the matrix system of equations to solve for optimal weight allocations of securities, given a set of constraints. As the number of securities in the portfolio increases, the size of matrix A increases and it becomes computationally expensive to compute the matrix inversion of A. Thus, one may consider methods such as the Cholesky decomposition, LU decomposition, QR decomposition, the Jacobi method, or the Gauss-Seidel method to break down matrix A into simpler matrices for factorization.

The LU decomposition

The **LU decomposition**, or also known as **lower upper factorization**, is one of the methods of solving square systems of linear equations. As its name implies, the LU factorization decomposes matrix A into a product of two matrices: a lower triangular matrix L and an upper triangular matrix U. The decomposition can be represented as follows:

$$A = LU$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{32} \end{bmatrix} \times \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Here, we can see $a = l_{11}u_{11}$, $b = l_{11}u_{12}$, and so on. A lower triangular matrix is a matrix that contains values in its lower triangle with the remaining upper triangle populated with zeros. The converse is true for an upper triangular matrix.

The definite advantage of the LU decomposition method over the Cholesky decomposition method is that it works for any square matrices. The latter method only works for symmetric and positive definite matrices.

Remember the previous example of a 3 by 3 matrix A. This time, though, we will use the `linalg` package of the `SciPy` module for the LU decomposition:

```
""" LU decomposition with SciPy """
import scipy.linalg as linalg
import numpy as np

A = np.array([[2., 1., 1.],
              [1., 3., 2.],
              [1., 0., 0.]])
B = np.array([4., 5., 6.])

LU = linalg.lu_factor(A)
x = linalg.lu_solve(LU, B)
```

To view the values of *x*, execute the following command:

```
>>> print x
[  6.  15. -23.]
```

We get the same values of 6, 15, and -23 for *a*, *b*, and *c* respectively.

Note that we used the `lu_factor` function of `scipy.linalg` here, which gives the `LU` variable as the pivoted LU decomposition of matrix *A*. We used the `lu_solve` function that takes in the pivoted LU decomposition and the vector *B* to solve the equation system.

We can display the LU decomposition of matrix *A* using the `lu` function. The `lu` function returns three variables : the permutation matrix *P*, the lower triangular matrix *L*, and the upper triangular matrix *U*, individually:

```
>>> P, L, U = scipy.linalg.lu(A)
```

When we print out these variables, we can conclude the relationship between the LU factorization and matrix *A* as follows:

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 2 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & -0.2 & 1 \end{bmatrix} \times \begin{bmatrix} 2 & 1 & 1 \\ 0 & 2.5 & 1.5 \\ 0 & 0 & -0.2 \end{bmatrix}$$

The LU decomposition can be viewed as the matrix form of Gaussian elimination performed on two simpler matrices: the upper triangular and lower triangular matrices.

The Cholesky decomposition

The Cholesky decomposition is another way of solving systems of linear equations. It can be significantly faster and uses a lot of less memory than the LU decomposition by exploiting the property of symmetric matrices. However, it is required that the matrix being decomposed be Hermitian (or real-valued symmetric and thus square) and positive definite. This means that when the matrix *A* is decomposed as $A = LL^T$, *L* is a lower triangular matrix with real and positive numbers on the diagonals, and L^T is the conjugate transpose of *L*.

Let's consider another example of a system of linear equations where matrix *A* is both Hermitian and positive definite. Again, the equation is in the form of $Ax = B$, where *A* and *B* take the following values:

$$A = \begin{bmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & -1 \\ 0 & 3 & -1 & 8 \end{bmatrix}, x = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}, B = \begin{bmatrix} 6 \\ 25 \\ -11 \\ 15 \end{bmatrix}$$

Let's represent these matrices as the `NumPy` arrays:

```
""" Cholesky decomposition with NumPy """
import numpy as np
```

```
A = np.array([[10., -1., 2., 0.],
              [-1., 11., -1., 3.],
              [2., -1., 10., -1.],
              [0.0, 3., -1., 8.]])
B = np.array([6., 25., -11., 15.])

L = np.linalg.cholesky(A)
```

The `cholesky` function of `numpy.linalg` would compute the lower triangular factor of matrix *A*. Let's view the lower triangular matrix:

```
>>> print L
[[ 3.16227766  0.          0.          0.        ]
 [-0.31622777  3.3015148   0.          0.        ]
 [ 0.63245553 -0.24231301  3.08889696  0.        ]
 [ 0.          0.9086738  -0.25245792  2.6665665 ]]
```

To verify that the Cholesky decomposition results are correct, we can use the definition of the Cholesky factorization by multiplying *L* with its conjugate transpose that will lead us back to the values of matrix *A*:

```
>>> print np.dot(L, L.T.conj())  # A=L.L*
[[ 10.  -1.   2.   0.]
 [ -1.  11.  -1.   3.]
 [  2.  -1.  10.  -1.]
 [  0.   3.  -1.   8.]]
```

Before solving for x, we need to solve for $L^T x$ as y. We will use the `solve` function of `numpy.linalg`:

```
>>> y = np.linalg.solve(L, B)  # L.L*.x=B; When L*.x=y, then L.y=B
```

To solve for x, all we need to do is to solve again using the conjugate transpose of *L* and *y*:

```
>>> x = np.linalg.solve(L.T.conj(), y)  # x=L*'.y
```

Let's print our result of x:

```
>>> print x
[ 1.  2. -1.  1.]
```

The output gives us our values of x for a, b, c, and d.

To show that the Cholesky factorization gives us the correct values, we can verify the answer by multiplying the matrix A by the transpose of x to return the values of B:

```
>>> print np.mat(A) * np.mat(x).T  # B=Ax
[[  6.]
 [ 25.]
 [-11.]
 [ 15.]]
```

This shows that the values of x by the Cholesky decomposition would lead to the same values given by B.

The QR decomposition

The QR decomposition, also known as the QR factorization, is another method of solving linear systems of equations using matrices, very much like the LU decomposition. The equation to solve is in the form of $Ax = B$, where matrix $A = QR$. Except in this case, A is a product of an orthogonal matrix Q and upper triangular matrix R. The QR algorithm is commonly used to solve the linear least squares problem.

An orthogonal matrix exhibits the following properties:

- It is a square matrix
- Multiplying an orthogonal matrix by its transpose returns the identity matrix:

$$QQ^T = Q^T Q = 1$$

- The inverse of an orthogonal matrix equals its transpose:

$$Q^T = Q^{-1}$$

An identity matrix is also a square matrix with its main diagonal containing ones and zeros elsewhere.

We can now restate the problem $Ax = B$ as follows:

$$QRx = B$$

$$Rx = Q^{-1}B \text{ or } Rx = Q^{T}B$$

Using the same variables in the LU decomposition example, we will use the `qr` function of `scipy.linalg` to compute our values of Q and R, and let the variable y represent our value of BQ^T with the following code:

```
""" QR decomposition with scipy """
import scipy.linalg as linalg
import numpy as np

A = np.array([
    [2., 1., 1.],
    [1., 3., 2.],
    [1., 0., 0]])
B = np.array([4., 5., 6.])

Q, R = scipy.linalg.qr(A)  # QR decomposition
y = np.dot(Q.T, B)  # Let y=Q'.B
x = scipy.linalg.solve(R, y)  # Solve Rx=y
```

Note that `Q.T` is simply the transpose of Q, which is also the same as the inverse of Q.

```
>>> print x
[  6.  15. -23.]
```

We get the same answers similar to those in the LU decomposition example.

Solving with other matrix algebra methods

So far, we have looked at the use of matrix inversion, the LU decomposition, the Cholesky decomposition, and QR decomposition to solve for systems of linear equations. Should the size of our financial data in matrix A be large, it can be broken down by a number of schemes so that the solution can converge more quickly using matrix algebra. Quantitative portfolio analysts ought to be familiar with these discussed methods.

In some circumstances, the solution that we are looking for might not converge. Therefore, one might consider the use of iterative methods. The common methods of solving systems of linear equations iteratively are the Jacobi method, the Gauss-Seidel method, and the SOR method. We will take a brief look at the examples in implementing the Jacobi and the Gauss-Seidel method.

The Jacobi method

The Jacobi method solves a system of linear equations iteratively along its diagonal elements. The iteration procedure terminates when the solution converges. Again, the equation to solve is in the form of $Ax = B$, where A can be decomposed into a diagonal component D and remainder R such that $A = D + R$. Let's take a look at the example of a 4 by 4 matrix A:

$$A = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & k & 0 \\ 0 & 0 & 0 & p \end{bmatrix} + \begin{bmatrix} 0 & b & c & d \\ e & 0 & g & h \\ i & j & 0 & l \\ m & n & o & 0 \end{bmatrix}$$

The solution is then obtained iteratively as follows:

$$Ax = B$$

$$\left(D + R\right)x = B$$

$$Dx = B - Rx$$

$$x_{n+1} = D^{1}\left(B - Rx_{n}\right)$$

As opposed to the Gauss-Siedel method, the value of x_n in the Jacobi method is needed during each iteration in order to compute x_{n+1} and cannot be overwritten. This would take up twice the amount of storage. However, the computations for each element can be done in parallel, which is useful for faster computations.

If matrix A is strictly irreducibly diagonally dominant, this method is guaranteed to converge. A strictly irreducibly diagonally dominant matrix is one where the absolute diagonal element in every row is greater than the sum of the absolute values of other terms.

In some circumstances, the Jacobi method can converge even if these conditions are not met. The Python code is given as follows:

```
""" Solve Ax=B with the Jacobi method """
import numpy as np


def jacobi(A, B, n, tol=1e-10):
    # Initializes x with zeroes with same shape and type as B
    x = np.zeros_like(B)

    for it_count in range(n):
        x_new = np.zeros_like(x)
        for i in range(A.shape[0]):
            s1 = np.dot(A[i, :i], x[:i])
            s2 = np.dot(A[i, i + 1:], x[i + 1:])
            x_new[i] = (B[i] - s1 - s2) / A[i, i]

        if np.allclose(x, x_new, tol):
            break

        x = x_new

    return x
```

Consider the same matrix values in the Cholesky decomposition example. We will use 25 iterations in our `jacobi` function to find the values of x.

```
A = np.array([[10., -1., 2., 0.],
              [-1., 11., -1., 3.],
              [2., -1., 10., -1.],
              [0.0, 3., -1., 8.]])
B = np.array([6., 25., -11., 15.])
n = 25
```

After initializing the values, we can now call the function and solve for x:

```
>>> x = jacobi(A, B, n)
>>> print "x =", x
x = [ 1.  2. -1.  1.]
```

We solved for the values of x, which are similar to the answers from the Cholesky decomposition.

The Gauss-Seidel method

The Gauss-Seidel method works very much like the Jacobi method. It is another way of solving a square system of linear equations using an iterative procedure with the equation in the form of $Ax = B$. Here, the matrix A is decomposed as $A = L + U$, where the matrix A is a sum of a lower triangular matrix L and an upper triangular matrix U. Let's take a look at the example of a 4 by 4 matrix A:

$$A = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} = \begin{bmatrix} a & 0 & 0 & 0 \\ e & f & 0 & 0 \\ i & j & k & 0 \\ m & n & o & p \end{bmatrix} + \begin{bmatrix} 0 & b & c & d \\ 0 & 0 & g & h \\ 0 & 0 & 0 & l \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The solution is then obtained iteratively as follows:

$$Ax = B$$

$$\left(L + U\right)x = B$$

$$Lx = B - Ux$$

$$x_{n+1} = L^{-1}\left(B - Ux_n\right)$$

Using a lower triangular matrix L, where zeroes fill up the upper triangle, the elements of x_n can be overwritten in each iteration in order to compute x_{n+1}. This results in the advantage of needing half the storage required when using the Jacobi method.

The rate of convergence in the Gauss-Seidel method largely lies in the properties of matrix A, especially if matrix A is needed to be strictly diagonally dominant or symmetric positive definite. Even if these conditions are not met, the Gauss-Seidel method may converge.

The Python implementation of the Gauss-Seidel method is given as follows:

```
""" Solve Ax=B with the Gauss-Seidel method """
import numpy as np


def gauss(A, B, n, tol=1e-10):
```

```
    L = np.tril(A)  # Returns the lower triangular matrix of A
    U = A - L  # Decompose A = L + U
    L_inv = np.linalg.inv(L)
    x = np.zeros_like(B)

    for i in range(n):
        Ux = np.dot(U, x)
        x_new = np.dot(L_inv, B - Ux)

        if np.allclose(x, x_new, tol):
            break

        x = x_new

    return x
```

Here, the `tril` function of NumPy returns the lower triangular matrix A from which we can find the lower triangular matrix U. Plugging the remaining values into x iteratively would lead us to the solution below with some tolerance defined by `tol`.

Let's consider the same matrix values in the Jacobi method and Cholesky decomposition example. We will use a maximum of 100 iterations in our `guass` function to find the values of x as follows:

```
A = np.array([[10., -1., 2., 0.],
              [-1., 11., -1., 3.],
              [2., -1., 10., -1.],
              [0.0, 3., -1., 8.]])
B = np.array([6., 25., -11., 15.])
n = 100
x = gauss(A, B, n)
```

Let's see if our x values match with those as before:

```
>>>print "x =", x
x = [ 1.  2. -1.  1.]
```

We solved for the values of x, which are similar to the answers from the Jacobi method and Cholesky decomposition.

Summary

In this chapter, we took a brief look at the use of the CAPM model and APT model in finance. In the CAPM model, we visited the efficient frontier with the capital market line to determine the optimal portfolio and the market portfolio. Then, we solved for the security market line using regression that helped us to determine whether an asset is undervalued or overvalued. In the APT model, we explored how various factors affect security returns other than using the mean-variance framework. We performed a multivariate regression to help us determine the coefficients of these factors that led to the valuation of our security price.

In portfolio allocation, portfolio managers are typically mandated by investors to achieve a set of objectives while following certain constraints. We can model this problem using linear programming. Using the PuLP Python package, we defined a maximization or minimization objective function, and added inequality constraints to our problems to solve for unknown variables. The three outcomes in linear optimization can either be an unbounded solution, only one solution, or no solution at all.

Another form of linear optimization is integer programming, where all the variables are restricted to be integers instead of fractional values. A special case of an integer variable is a binary variable, which can either be 0 or 1, and it is especially useful to model decision making given a set of choices. We worked on a simple integer programming model containing binary conditions and saw how easy it is to run into a pitfall. Careful planning on the design of integer programming models is required for them to be useful in decision making.

The portfolio allocation problem may also be represented as a system of linear equations with equalities, which can be solved using matrices in the form of $Ax = B$. To find the values of x, we solved for $A^{-1}B$ using various types of decomposition of the matrix A. The two types of matrix decomposition methods are the direct and indirect methods. The direct method performs matrix algebra in a fixed number of iterations. They are namely the LU decomposition, Cholesky decomposition, and QR decomposition methods. The indirect or iterative method iteratively computes the next values of x until a certain tolerance of accuracy is reached. This method is particularly useful for computing large matrices, but it also faces the risk of not having the solution converge. The indirect methods we have used are the Jacobi method and the Gauss-Seidel method.

In the next chapter, we will take a look at nonlinear models and methods of solving them.

3
Nonlinearity in Finance

In recent years, there has been a growing interest in research on nonlinear phenomena in economic and financial theory. With nonlinear serial dependence playing a significant role in the returns of many financial time series, this makes security valuation and pricing very important, leading to an increase in studies of nonlinear modeling of financial products.

Practitioners in the financial industry use nonlinear models to forecast volatility, price derivatives, and compute **Value at Risk** (**VAR**). Unlike linear models, where linear algebra is used to find a solution, nonlinear models do not necessarily infer a global optimal solution. Numerical root-finding methods are usually employed to converge toward the nearest local optimal solution, which is a root.

In this chapter, we will discuss the following topics to explore some methods that will help us extract information from nonlinear models:

- Examining the definition of nonlinearity
- Discussing the volatility smile in implied volatility modeling
- Discussing Markov switching models, threshold models, and smooth transition models as nonlinear models
- An overview of root-finding to find the optimal point of nonlinear models
- Examining the incremental search algorithm, bisection algorithm, Newton's algorithm, and secant method in root-finding
- Combining root-finding methods with Brent's method
- SciPy's implementation of root-finding methods as scalar functions
- SciPy's general nonlinear solvers in root-finding

Nonlinearity modeling

While linear relationships aim to explain observed phenomena in the simplest way possible, many complex physical phenomena cannot be explained using such models. A nonlinear relationship is defined as follows:

$$f(a+b) \neq f(a)+f(b)$$

Even though nonlinear relationships may be complex, to fully understand and model them we will take a look at the examples that are applied in the context of finance and in time series models.

Examples of nonlinear models

Many nonlinear models have been proposed for academic and applied research to explain certain aspects of economic and financial data that are left unexplained by linear models. The literature on nonlinearity in finance is simply too broad and deep to be adequately explained in this book. In this section, we will just briefly discuss some examples of nonlinear models that we may possibly come across for practical uses: the implied volatility model, Markov-switching model, threshold model, and smooth transition model.

The implied volatility model

Perhaps one of the most widely studied option pricing models is the Black-Scholes-Merton model, or simply the Black-Scholes model in short. A call (put) option is a right, not an obligation, to buy (sell) the underlying security at a particular price and at a particular time. The Black-Scholes model helps determine the fair price of an option with the assumption that returns of the underlying security are normally distributed (N(.)) or that asset prices are log-normally distributed.

The formula takes on the following assumed variables: the strike price (K), time to expiry (T), risk-free rate (r), volatility of the underlying returns (σ), current price of the underlying asset (S), and its yield (q). The mathematical formula for a call option $C(S,t)$ is represented as follows:

$$C(S,t) = Se^{-qT}N(d_1) - Ke^{-rT}N(d_2)$$

Here:

$$d_1 = \frac{ln\left(S/K\right) + \left(r - q + \sigma^2/2\right)T}{\sigma\sqrt{T}}$$

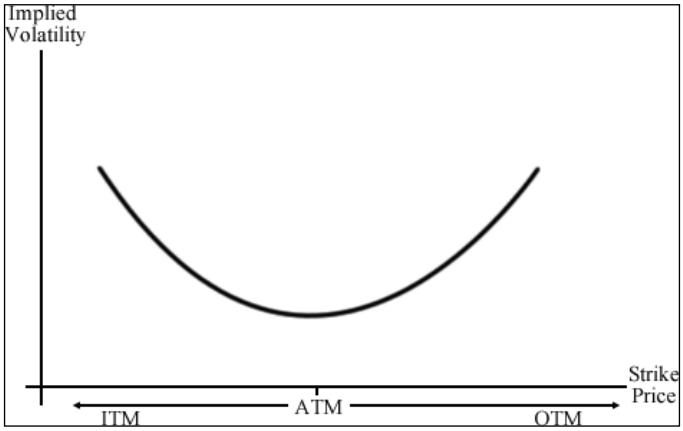
$$d_2 = d_1 - \sigma\sqrt{T}$$

By way of market forces, the price of an option may deviate from the price derived from the Black-Scholes formula. In particular, the realized volatility (that is, the observed volatility of the underlying returns from historical market prices) could differ from the volatility value as implied by the Black-Scholes model, which is indicated by σ.

Remember the CAPM model discussed in *Chapter 2, The Importance of Linearity in Finance*. In general, securities that have higher returns exhibit higher risk, as indicated by the volatility or standard deviation of returns.

With volatility being such an important factor in security pricing, many volatility models have been proposed for studies. One such model is the implied volatility modeling of option prices.

Suppose we plot the implied volatility values of an equity option given by the Black-Scholes formula with a particular maturity for every strike price available. In general, we get a curve commonly known as the *volatility smile* due to its shape:



The implied volatility typically is its highest for **deep in-the-money** (**ITM**) and **out-of-the-money** (**OTM**) options driven by heavy speculation and at its lowest for **at-the-money** (**ATM**) options.

The characteristics of options are explained as follows:

- **In-the-money options (ITM)**: A call option is considered ITM when its strike price is below the market price of the underlying asset. A put option is considered ITM when its strike price is above the market price of the underlying asset. ITM options have an intrinsic value when exercised.
- **Out-of-the-money options (OTM)**: A call option is considered OTM when its strike price is above the market price of the underlying asset. A put option is considered ITM when its strike price is below the market price of the underlying asset. OTM options have no intrinsic value when exercised.
- **At-the-money options (ATM)**: An option is considered ATM when its strike price is the same as the market price of the underlying asset. ATM options have no intrinsic value, but may still have time value.

From the preceding volatility curve, one of the objectives in implied volatility modeling is to seek the lowest implied volatility value possible or, in other words, to "find the root". When found, the theoretical price of an ATM option for a particular maturity can be deduced and compared against the market prices for potential opportunities, such as for studying near ATM options or far OTM options. However, since the curve is nonlinear, linear algebra cannot adequately solve for the root. We will take a look at a number of root-finding methods in the later sections of this chapter.

The Markov regime-switching model

To model nonlinear behavior in economic and financial time series, Markov switching models can be used to characterize time series in different states of the world or regimes. Examples of such states could be a *volatile* state as seen in the 2008 global economic downturn, or a *growth* state of a steadily recovering economy. The ability to transit between these structures lets the model capture complex dynamic patterns.

The Markov property of stock prices implies that only the present values are relevant for predicting the future. Past stock price movements are irrelevant to the way the present has emerged.

Let's take an example of a Markov regime-switching model with $m = 2$ regimes:

$$y_t \begin{cases} x_1 + \varepsilon_t, when\, s_t = 1 \\ x_2 + \varepsilon_t, when\, s_t = 2 \end{cases}$$

ε_t is an independent and **identically distributed** (**i.i.d**) white noise. White noise is a normally distributed random process with a mean of zero. The same model can be represented with dummy variables:

$$y_t = x_1 D_t + x_2 \left(1 - D_t\right) + \varepsilon_t$$

$$where\, D_t = 1\, when\, s_t = 1,$$

$$or\, D_t = 0\, when\, s_t = 2$$

The application of Markov switching models includes representing the real GDP growth rate and inflation rate dynamics. These models in turn drive the valuation models of interest-rate derivatives. The probability of switching from the previous state i to the current state j can be written as follows:

$$P\left[s_t = j \mid s_{t-1} = i\right]$$

The threshold autoregressive model

One popular class of nonlinear time series models is the **threshold autoregressive** (**TAR**) model, which looks very similar to the Markov switching models. Using regression methods, simple AR models are arguably the most popular models to explain nonlinear behavior. Regimes in the threshold model are determined by past d values of its own time series, relative to a threshold value c. The following is an example of a self-exciting TAR (SETAR) model. The SETAR model is self-exciting because switching between different regimes depends on the past values of its own time series:

$$y_t \begin{cases} a_1 + b_1 y_{t-d} + \varepsilon_t, if\ y_{t-d} \le c \\ a_2 + b_2 y_{t-d} + \varepsilon_t, if\ y_{t-d} > c \end{cases}$$

Using dummy variables, the SETAR model can also be represented as follows:

$$y_t = \left(a_1 + b_1 y_{t-d}\right) D_t + \left(a_2 + b_2 y_{t-d}\right)\left(1 - D_t\right) + \varepsilon_t$$

$$where\, D_t = 1\, when\, y_{t-d} \le c,$$

$$or\, D_t = 0\, when\, y_{t-d} > c$$

Note that the use of the TAR model may result in sharp transitions between the states as controlled by the threshold variable *c*.

Smooth transition models

Abrupt regime changes in the threshold models appear to be unrealistic against real-world dynamics. This problem can be overcome by introducing a smoothly changing continuous function from one regime to another. The SETAR model becomes a logistic smooth transition threshold autoregressive (LSTAR) model with the logistic function $G\left(y_{t-1};\gamma,c\right)$:

$$G\left(y_{t-1};\gamma,c\right) = \frac{1}{1+e^{-\gamma\left(y_{t-d}-c\right)}}$$

The SETAR model now becomes a LSTAR model, as shown in the following equation:

$$y_t = \left(a_1 + b_1 y_{t-d}\right)\left(1 - G\left(y_{t-1};\gamma,c\right)\right) + \left(a_2 + b_2 y_{t-d}\right) G\left(y_{t-1};\gamma,c\right) + \varepsilon_t$$

The parameter γ controls the smooth transition from one regime to another. For large values of γ, the transition is the fastest, as y_{t-d} approaches the threshold variable *c*. When $\gamma = 0$, the LSTAR model is equivalent to a simple AR(1) one-regime model.

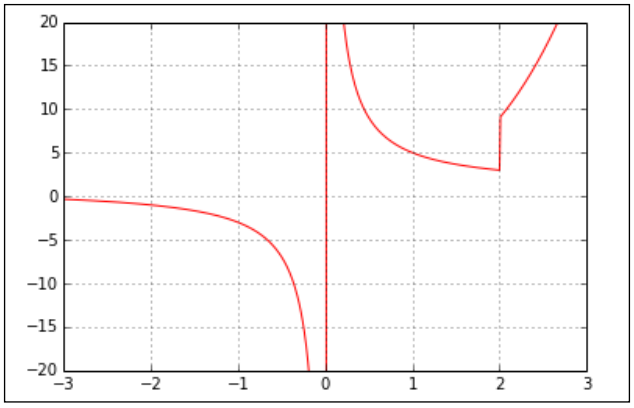
An introduction to root-finding

In the preceding section, we discussed some nonlinear models commonly used for studying economics and financial time series. From the model data given in continuous time, the intention is therefore to search for the extrema that could possibly infer valuable information. The use of numerical methods, such as root-finding algorithms, can help us find the roots of a continuous function f such that $f(x) = 0$, which can either be the maxima or the minima of the function. In general, an equation may either contain a number of roots or none at all.

One example of the use of root-finding methods on nonlinear models is the Black-Scholes implied volatility modeling discussed earlier. An option trader would be interested in deriving implied prices based on the Black-Scholes model and comparing them with the market prices. In the next chapter, we will see how we can combine a root-finding method with a numerical option pricing procedure to create an implied volatility model based on the market prices of a particular option.

Root-finding methods use an iterative routine that requires a start point or the estimation of the root. The estimation of the root can either converge toward a solution, converge to a root that is not sought, or may not even find a solution at all. Thus, it is crucial to find a good approximation to the root.

Not every nonlinear function can be solved using root-finding methods. The following figure shows an example of a continuous function where root-finding methods may fail to arrive at a solution. There are discontinuities at $x = 0$ and $x = 2$ for the y values in the range of -20 to 20:

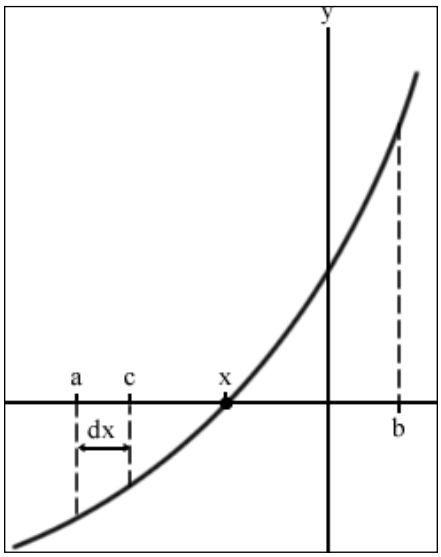


There is no fixed rule as to how a good approximation can be defined. It is recommended that you bracket or define the lower and upper search bounds before starting the root-finding iterative procedure. We certainly do not want to keep searching in the wrong direction endlessly for our root.

Incremental search

A crude method of solving a nonlinear function is by doing an incremental search. Using an arbitrarily starting point a, we can obtain values of $f(a)$ for every increment of dx. We assume that the values of $F(a+dx)$, $f(a+2dx)$, $f(a+3dx)$… are going in the same direction as indicated by their sign. Once the sign changes, a solution is deemed as found. Otherwise, the iterative search terminates when it crosses the boundary point b.

A pictorial example of the root-finder method for iteration is given in the following graph:



An example can be seen from the Python code:

```
'''
Python code:
Incremental search method
'''
""" An incremental search algorithm """
import numpy as np


def incremental_search(f, a, b, dx):
    """
    :param f: The function to solve
    :param a: The left boundary x-axis value
    :param b: The right boundary x-axis value
```

```
    :param dx: The incremental value in searching
    :return: The x-axis value of the root,
                number of iterations used
    """
    fa = f(a)
    c = a + dx
    fc = f(c)
    n = 1

    while np.sign(fa) == np.sign(fc):
        if a >= b:
            return a - dx, n

        a = c
        fa = fc
        c = a + dx
        fc = f(c)
        n += 1

    if fa == 0:
        return a, n
    elif fc == 0:
        return c, n
    else:
        return (a + c)/2., n
```

At every iterative procedure, a will be replaced by c, and c will be incremented by dx before the next comparison. Should a root be found, it is plausible that it lies between a and c, both inclusive. In the event should the solution not rest at either point, we will simply return the average of the two points as the best estimation. The variable n keeps track of the number of iterations that underwent the process of finding our root.

We will use the equation that has an analytic solution of $y = x^3 + 2x^2 - 5$ to demonstrate and measure our root-finder, where x is bounded between -5 and 5. A small dx value of 0.001 is given, which also acts as a precision tool. Smaller values of dx produce better precision but also require more search iterations:

```
>>> """
>>> The keyword 'lambda' creates an anonymous function
>>> with input argument x
>>> """
>>> y = lambda x: x**3 + 2.0*x**2 - 5.
>>> root, iterations = incremental_search (y, -5., 5., 0.001)
>>> print "Root is:", root
>>> print "Iterations: ", iterations
```

```
Root is: 1.2415
Iterations:  6242
```

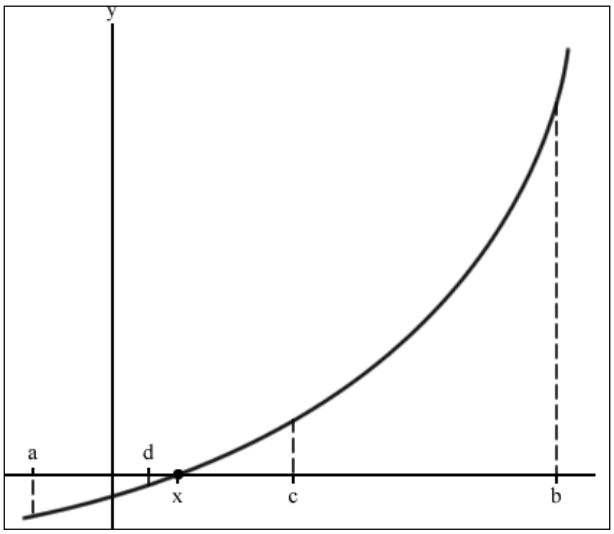
The incremental search root-finder method is a basic demonstration of the fundamental behavior of a root-finding algorithm. The accuracy is at its best when defined by *dx* and consumes an extremely long computational time in the worst possible scenario. The higher the accuracy demanded, the longer it takes for the solution to converge. For practical reasons, this method is the least preferred of all root-finding algorithms, and we will take a look at alternative methods to find the roots of our equation, which can give us a better performance.

The bisection method

The bisection method is considered the simplest one-dimensional root-finding algorithm. The general interest is to find the value x of a continuous function f such that $f(x) = 0$.

Suppose we know the two points of an interval a and b, where $a < b$, and that $f(a) < 0$ and $f(b) > 0$ lie along the continuous function, taking the midpoint of this interval as c, where $c = \frac{a+b}{2}$, the bisection method then evaluates this value as *f(c)*.

Let's illustrate the setup of points along a nonlinear function with the following graph:



Since the value of *f(a)* is negative and *f(b)* is positive, the bisection method assumes that the root *x* lies somewhere between *a* and *b* and gives $f(x) = 0$.

If $f(c) = 0$ or is very close to zero by some predetermined error tolerance value, then a root is declared as found. If $f(c) < 0$, then we may conclude that a root exists along the interval c and b, or interval a and c otherwise.

On the next evaluation, c is replaced as either a or b accordingly. With the new interval shortened, the bisection method repeats with the same evaluation to determine the next value of c. This process continues, shrinking the width of the interval ab until the root is determined as found.

The biggest advantage of using the bisection method is its guarantee to converge to an approximation of the root, given a predetermined error tolerance level and the maximum number of iterations allowed. It should be noted that the bisection method does not require knowledge of the derivative of the unknown function. In certain continuous functions, the derivative could be complex or even impossible to calculate. This makes the bisection method extremely valuable for working on functions that are not smooth.

Because the bisection method does not require derivative information from the continuous function, its major drawback is that it takes up more computational time in the iterative evaluation as compared to other root-finder methods. Also, since the search boundary of the bisection method lies in the intervals a and b, it would require a good approximation to ensure that the root falls within this range. Otherwise, a wrong solution may be obtained or even none at all. Using large values of a and b might consume more computational time.

The bisection is considered to be stable without the use of an initial guess value for convergence to happen. Often, it is used in combination with other methods, such as the faster Newton's method, to converge quickly with precision.

Save this file as `bisection.py`. The Python code for the bisection method is given as follows:

```
""" The bisection method """


def bisection(f, a, b, tol=0.1, maxiter=10):
    """
    :param f: The function to solve
    :param a: The x-axis value where f(a)<0
    :param b: The x-axis value where f(b)>0
    :param tol: The precision of the solution
```

```
        :param maxiter: Maximum number of iterations
        :return: The x-axis value of the root,
                    number of iterations used
        """
        c = (a+b)*0.5  # Declare c as the midpoint ab
        n = 1  # Start with 1 iteration
        while n <= maxiter:
            c = (a+b)*0.5
            if f(c) == 0 or abs(a-b)*0.5 < tol:
                # Root is found or is very close
                return c, n

            n += 1
            if f(c) < 0:
                a = c
            else:
                b = c

        return c, n
```

Let's try out our bisection method:

```
>>> y = lambda x: x**3 + 2*x**2 - 5
>>> root, iterations = bisection(y, -5, 5, 0.00001, 100)
>>> print "Root is:", root
>>> print "Iterations: ", iterations
Root is: 1.24190330505
Iterations:  20
```

Again, we bounded the anonymous function lambda to the variable y with an input parameter x and attempted to solve the equation $y = x^3 + 2x^2 - 5$ as before in the interval between -5 to 5 to an accuracy of 0.00001 with a maximum iteration of 100.

As we can see, the result from the bisection method gives us better precision in far fewer iterations over the incremental search method.

Newton's method

Newton's method, also known as the Newton-Raphson method, uses an iterative procedure to solve for a root using information about the derivative of a function. The derivative is treated as a linear problem to be solved. The first-order derivation f' of the function f represents the tangent line. The approximation to the next value of x, given as x_1, is as follows:

$$x_1 = x - \frac{f(x)}{f'(x)}$$

Here, the tangent line intersects the x axis at x_1, which produces $y = 0$. This also represents the first-order Taylor expansion about x_1 such that that the new point $x_1 = x + \Delta x$ solves the following equation:

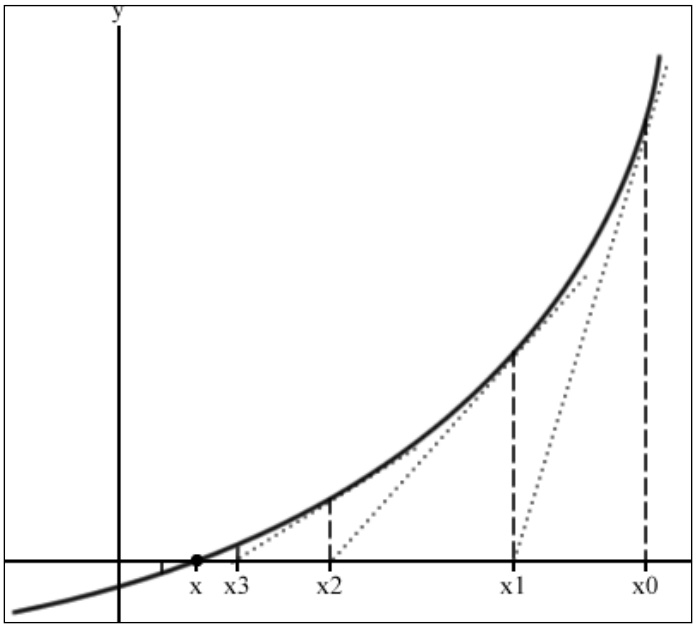
$$f(x_1 + \Delta x) = 0$$

This process is repeated with x taking the value of x_1 until the maximum number of iterations is reached, or the absolute difference between x_1 and x is within an acceptable accuracy level.

An initial guess value is required to compute the values of $f(x)$ and $f'(x)$. The rate of convergence is quadratic, which is considered to be extremely fast in obtaining the solution with high levels of accuracy.

The drawback to Newton's method is that it does not guarantee global convergence to the solution. Such a situation arises when the function contains more than one root, or when the algorithm arrives at a local extremum and is unable to compute the next step. As this method requires knowledge of the derivative of its input function, it is required that the input function be differentiable. However, in certain circumstances, it is impossible for the derivative of a function to be known, or otherwise be mathematically easy to compute.

A graphical representation of Newton's method is shown in the following screenshot. $x0$ is the initial x value. The derivative of $f(x0)$ is evaluated, which is a tangent line crossing the x axis at $x1$. The iteration is repeated, evaluating the derivative at points $x1$, $x2$, $x3$, and so on.



The implementation of Newton's method in Python is as follows:

```
""" The Newton-Raphson method """


def newton(f, df, x, tol=0.001, maxiter=100):
    """
    :param f: The function to solve
    :param df: The derivative function of f
    :param x: Initial guess value of x
    :param tol: The precision of the solution
    :param maxiter: Maximum number of iterations
    :return: The x-axis value of the root,
                number of iterations used
    """
    n = 1
```

```
        while n <= maxiter:
            x1 = x - f(x)/df(x)
            if abs(x1 - x) < tol:  # Root is very close
                return x1, n
            else:
                x = x1
                n += 1

        return None, n
```

We will use the same function used in the bisection example and take a look at the results from Newton's method:

```
>>> y = lambda x: x**3 + 2*x**2 - 5
>>> dy = lambda x: 3*x**2 + 4*x
>>> root, iterations = newton(y, dy, 5.0, 0.00001, 100)
>>> print "Root is:", root
>>> print "Iterations:", iterations
Root is: 1.24189656303
Iterations: 7
```

Beware of division by zero exceptions! Using values such as 5.0, instead of 5, lets Python recognize the variable as a float, avoids the problem of treating variables as integers in calculations, and gives us better precision.

With Newton's method, we obtained a really close solution with less iteration over the bisection method.

The secant method

The secant method uses secant lines to find the root. A secant line is a straight line that intersects two points of a curve. In the secant method, a line is drawn between two points on the continuous function such that it extends and intersects the x axis. This method can be thought of as a Quasi-Newton method. By successively drawing such secant lines, the root of the function can be approximated.

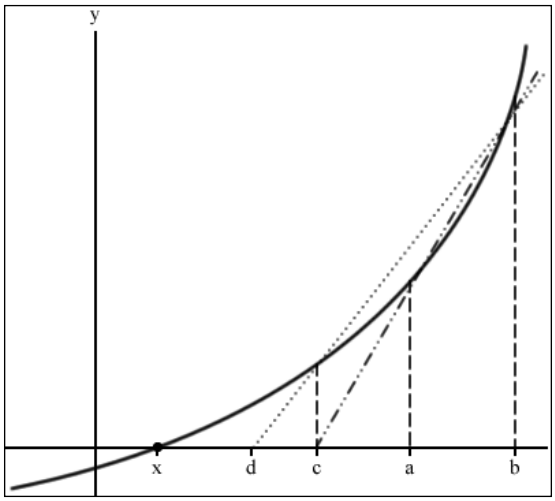
The secant method is graphically represented in the following screenshot. An initial guess of the two x axis values a and b is required to find $f(a)$ and $f(b)$. A secant line y is drawn from $f(b)$ to $f(a)$ and intersects at point c on the x axis such that:

$$y = \frac{f(b) - f(a)}{b - a}(c - b) + f(b)$$

The solution to c is therefore:

$$c = b - f(b)\frac{b - a}{f(b) - f(a)}$$

On the next iteration, a and b will take on the values of b and c respectively. The method repeats itself, drawing secant lines for the x axis values of a and b, b and c, c and d, and so on. The solution terminates when the maximum number of iterations has been reached or the difference between b and c has reached a prespecified tolerance level, as shown in the following graph:



The rate of convergence of the secant method is considered to be superlinear. Its secant method converges much faster than the bisection method and slower than Newton's method. In Newton's method, the number of floating-point operations takes up twice as much time as the secant method in the computation of both its function and its derivative on every iteration. Since the secant method requires only computation of its function at every step, it can be considered faster in absolute time.

It is required that the initial guess values of the secant method be close to the root, otherwise it has no guarantee of converging to the solution.

The Python code for the secant method is given as follows:

```
""" The secant root-finding method """


def secant(f, a, b, tol=0.001, maxiter=100):
    """
    :param f: The function to solve
    :param a: Initial x-axis guess value
    :param b: Initial x-axis guess value, where b>a
    :param tol: The precision of the solution
    :param maxiter: Maximum number of iterations
    :return: The x-axis value of the root,
                number of iterations used
    """
    n = 1
    while n <= maxiter:
        c = b - f(b)*((b-a)/(f(b)-f(a)))
        if abs(c-b) < tol:
            return c, n

        a = b
        b = c
        n += 1

    return None, n
```

Again, we will reuse the same nonlinear function and return the results from the secant method:

```
>>> y = lambda x: x**3 + 2*x**2 - 5
>>> root, iterations = secant(y, -5.0, 5.0, 0.00001, 100)
>>> print "Root is:", root
>>> print "Iterations:", iterations
Root is: 1.24189656226
Iterations: 14
```

Though all of the preceding root-finding methods gave very close solutions, the secant method performs with fewer iterations compared to the bisection method, but with more than Newton's method.

Combining root-finding methods

It is perfectly possible to write your own root-finding algorithms using a combination of the previously mentioned root-finding methods. For example, you may use the following implementation in the following order:

1. Use the faster secant method to converge the problem to a prespecified error tolerance value or a maximum number of iterations.
2. Once a prespecified tolerance level is reached, switch to using the bisection method to converge to the root by halving the search interval with each iteration until the root is found.

Brent's method or the **Wijngaarden-Dekker-Brent** method combines the bisection root-finding method, secant method, and inverse quadratic interpolation. The algorithm attempts to use either the secant method or inverse quadratic interpolation whenever possible, and uses the bisection method where necessary.

Brent's method can also be found in the `scipy.optimize.brentq` function of SciPy.

SciPy implementations

Before starting on writing your root-finding algorithm to solve nonlinear or even linear problems, take a look at the documentation of the `scipy.optimize` methods. SciPy contains a collection of scientific computing functions as an extension of Python. Chances are that these open source algorithms might fit into your applications off-the-shelf.

Root-finding scalar functions

Some root-finding functions that can be found in the `scipy.optimize` modules are `bisect`, `newton`, `brentq`, and `ridder`. Let's set up the examples that we have discussed using the implementations by SciPy:

```
"""
Documentation at
http://docs.scipy.org/doc/scipy/reference/optimize.html
"""
import scipy.optimize as optimize

y = lambda x: x**3 + 2.*x**2 - 5.
dy = lambda x: 3.*x**2 + 4.*x

# Call method: bisect(f, a, b[, args, xtol, rtol, maxiter, ...])
print "Bisection method: %s" \
      % optimize.bisect(y, -5., 5., xtol=0.00001)

# Call method: newton(func, x0[, fprime, args, tol, ...])
print "Newton's method: %s" \
      % optimize.newton(y, 5., fprime=dy)
# When fprime=None, then the secant method is used.
print "Secant method: %s" \
      % optimize.newton(y, 5.)

# Call method: brentq(f, a, b[, args, xtol, rtol, maxiter, ...])
print "Brent's method: %s" \
      % optimize.brentq(y, -5., 5.)
```

When we run the preceding code, the following output is generated:

```
Bisection method: 1.24190330505
Newton's method: 1.24189656303
Secant method: 1.24189656303
Brent's method: 1.24189656303
```

We can see that the SciPy implementation gives us somewhat very close answers as our derived ones.

It should be noted that SciPy has a set of well-defined conditions for every implementation. For example, the function call of the bisection routine is given as follows:

```
scipy.optimize.bisect(f, a, b, args=(), xtol=1e-12,
rtol=4.4408920985006262e-16, maxiter=100, full_output=False,
disp=True)
```

The function will strictly evaluate the function `f` to return a zero of the function. $f(a)$ and $f(b)$ cannot have the same signs. In certain scenarios, it is difficult to fulfill these constraints. For example, in solving for nonlinear implied volatility models, volatility values cannot be negative. In active markets, finding a root or a zero of the volatility function is almost impossible without modifying the underlying implementation. In such cases, implementing our own root-finding methods might perhaps give us more authority over how our application should behave.

General nonlinear solvers

The `scipy.optimize` module also contains multidimensional general solvers that we can harness to our advantage. The `root` and `fsolve` functions are some examples with the following function properties:

- `root(fun, x0[, args, method, jac, tol, ...])`: This finds a root of a vector function
- `fsolve(func, x0[, args, fprime, ...])`: This finds the roots of a function

The outputs are returned as a dictionary object. Using our example again as inputs to these functions, we will get the following output:

```
>>> import scipy.optimize as optimize

>>> y = lambda x: x**3 + 2.*x**2 - 5.
>>> dy = lambda x: 3.*x**2 + 4.*x

>>> print optimize.fsolve(y, 5., fprime=dy)
[ 1.24189656]
```

```
>>> print optimize.root(y, 5.)
 status: 1
 success: True
     qtf: array([ -3.73605502e-09])
    nfev: 12
       r: array([-9.59451815])
     fun: array([  3.55271368e-15])
       x: array([ 1.24189656])
 message: 'The solution converged.'
    fjac: array([[-1.]])
```

Using an initial guess value of 5, our solution converged to the root at 1.24189656, which is pretty close to the answers we had so far. What happens when we choose a value on the other side of the graph? Let's use an initial guess value of -5:

```
>>> print optimize.fsolve(y, -5., fprime=dy)
[-1.33306553]
>>> print optimize.root(y, -5.)
  status: 5
 success: False
     qtf: array([ 3.81481521])
    nfev: 28
       r: array([-0.00461503])
     fun: array([-3.81481496])
       x: array([-1.33306551])
 message: 'The iteration is not making good progress, as measured by
the \n  improvement from the last ten iterations.'
    fjac: array([[-1.]])
```

As seen from the display output, the algorithms did not converge and return a root that is a little further away from our previous answers. If we take a look at the equation on a graph, there are a number of points along the curve that lie very close to the root. A root-finder would be needed to obtain the desired level of accuracy, while solvers attempt to solve for the nearest answer in the fastest time.

Summary

In this chapter, we briefly discussed the persistence of nonlinearity in economics and finance. We looked at some nonlinear models that are commonly used in finance to explain certain aspects of data left unexplained by linear models: the Black-Scholes implied volatility model, Markov switching model, threshold model, and smooth transition models.

In Black-Scholes implied volatility modeling, we discussed the volatility smile that was made up of implied volatilities derived via the Black-Scholes model from the market prices of call or put options for a particular maturity. You may be interested enough to seek the lowest implied volatility value possible, which can be useful for inferring theoretical prices and comparing against the market prices for potential opportunities. However, since the curve is nonlinear, linear algebra cannot adequately solve for the optimal point. To do so, we will require the use of root-finding methods.

Root-finding methods attempt to find the root of a function or its zero. We discussed common root-finding methods: the bisection method, Newton's method, and secant method. Using a combination of root-finding algorithms may help us to seek roots of complex functions faster. One such example is Brent's method.

We explored functionalities in the `scipy.optimize` module that contains these root-finding methods, albeit with constraints. One of these constraints requires that the two boundary input values be evaluated with a pair of a negative value and positive value for the solution to converge successfully. In implied volatility modeling, this evaluation is almost impossible since volatilities do not have negative values. Implementing our own root-finding methods might perhaps give us more authority over how our application should perform.

Using general solvers is another way of finding roots. They may also converge to our solution more quickly, but such a convergence is not guaranteed by the initial given values.

Nonlinear modeling and optimization are inherently a complex task, and there is no universal solution or a sure way to reach a conclusion. This chapter serves to introduce nonlinearity studies for finance in general.

In the next chapter, we will take a look at numerical methods commonly used for options pricing. By pairing a numerical procedure with a root-finding algorithm, we will learn how to build an implied volatility model with the market prices of an equity option.

4
Numerical Procedures

A derivative is a contract whose payoff depends on the value of some underlying asset. In cases where closed-form derivative pricing may be complex or even impossible, numerical procedures excel. A numerical procedure is the use of iterative computational methods in attempting to converge to a solution. One such basic implementation is a binomial tree. In a binomial tree, a node represents the state of an asset at a certain point of time associated with a price. Each node leads to two other nodes in the next time step. Similarly, in a trinomial tree, each node leads to three other nodes in the next time step. However, as the number of nodes or the time steps of trees increase, so do the computational resources consumed. Lattice pricing attempts to solve this problem by storing only the new information at each time step, while reusing values where possible.

In finite difference pricing, the nodes of the tree can also be represented as a grid. The terminal values on the grid consist of terminal conditions, while the edges of the grid represent boundary conditions in asset pricing. We will discuss the explicit method, implicit method, and the Crank-Nicolson method of the finite differences schemes to determine the price of an asset.

Although vanilla options and certain exotics such as European barrier options and lookback options can be found to have a closed-form solution, other exotic products such as Asian options do not contain a closed-form solution. In these cases, the pricing of options can be used with numerical procedures.

In this chapter, we will cover the following topics:

- Pricing European and American options using a binomial tree
- Using a Cox-Ross-Rubinstein (CRR) binomial tree
- Pricing options using a Leisen-Reimer (LR) tree

- Pricing options using a trinomial tree
- Pricing options using a binomial and trinomial lattice
- Deriving Greeks from a tree for free
- Finite differences with the explicit, implicit, and Crank-Nicolson method
- Implied volatility modelling using a LR tree and the bisection method

Introduction to options

An **option** is a derivative of an asset that gives an owner the right but not the obligation to transact the underlying asset at a certain date for a certain price, known as the maturity date and strike price respectively.

A **call option** gives the buyer the right to buy an asset by a certain date for a certain price. A seller or writer of a call option is obligated to sell the underlying security to the buyer at the agreed price, should the buyer exercise his/her rights on the agreed date. A **put option** gives the buyer the right to sell the underlying asset by a certain date for a certain price. A seller or writer of a put option is obligated to buy the underlying security from the buyer at the agreed price, should the buyer exercise his/her rights on the agreed date.

The most common options available are the European options and American options. Other exotic options include Bermudan options and Asian options. This chapter will deal mainly with European and American options. An European option can only be exercised on the maturity date. An American option may be exercised at any time throughout the lifetime of the option.

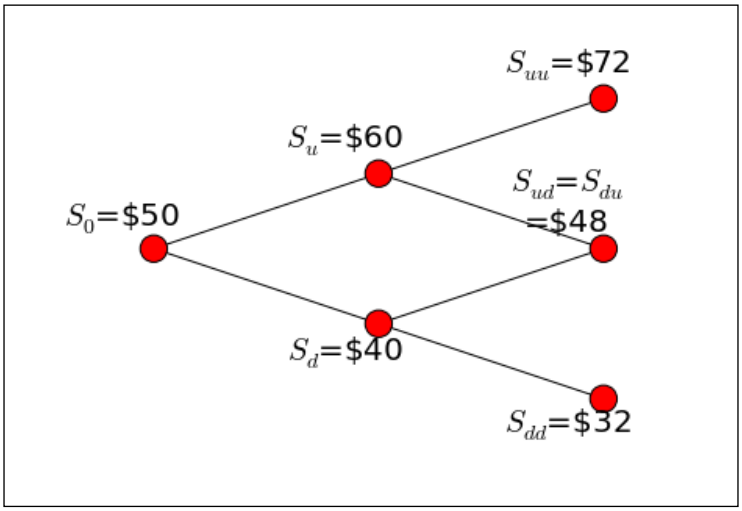
Binomial trees in options pricing

In the binomial options pricing model, the underlying security at one time period, represented as a node with a given price, is assumed to traverse to two other nodes in the next time step, representing an up state and a down state. Since options are derivatives of the underlying asset, the binomial pricing model tracks the underlying conditions on a discrete-time basis. Binomial option pricing can be used to value European options, American options, as well as Bermudan options.

The initial value of the root node is the spot price S_0 of the underlying security with a given probability of returns P_u should its value increase, and a probability of loss P_d should its value decrease. Based on these probabilities, the expected values of the security are calculated for each state of price increase or decrease for every time step. The terminal nodes represent every value of the expected security prices for every combination of up states and down states. We can then calculate the value of the option at every node, traverse the tree by risk-neutral expectations, and after discounting from the forward interest rates, we can derive the value of the call or put option.

Pricing European options

Consider a two-step binomial tree. A non-dividend paying stock price starts at $50, and in each of the two time steps, the stock may go up by 20 percent or go down by 20 percent. We suppose that the risk-free rate is 5 percent per annum and the time to maturity T is 0.5 years. We would like to find the value of an European put option with a strike price K of $52. The following figure shows the pricing of the stock using a binomial tree:



Here, the nodes are calculated as follows:

$$Spot\ Price, S_0 = 50$$

$$Probability\ of\ up\ state, p_u = 1.2$$

$$Probability\ of\ down\ state, p_d = 0.8$$

$$S_u = 50(1.2) = 60$$

$$S_d = 50(0.8) = 80$$

$$S_{uu} = 50(1.2)^2 = 72$$

$$S_{uu} = S_{ud} = 50(1.2)(0.8) = 48$$

$$S_{dd} = 50(0.8)^2 = 32$$

At the ultimate nodes, which hold the values of the underlying stock at maturity, the payoff from exercising an European call option is given as follows:

$$C_t = \max(0, S_t - K)$$

In the case of an European put option, the payoff is as follows:

$$p_t = \max(0, K - S_t)$$

From the option payoff values, we can then traverse the binomial tree backward to the current time and, after discounting from the risk-free rate, we will obtain our present value of the option. Traversing the tree backward takes into account the risk-neutral probabilities of the option's up states and down states.

We may assume that investors are indifferent to risk and that expected returns on all assets are equal. In the case of investing in stocks, by risk-neutral probability, the payoff from holding the stock, taking into account the up and down state possibilities, would be equal to the continuously compounded risk-free rate expected in the next time step, as follows:

$$e^{rt} = qu + (1-q)d$$

The risk-neutral probability q of investing in the stock can be rewritten as follows:

$$q = \frac{e^{rt} - d}{u - d}$$

Are these formulas relevant to stocks? What about futures?

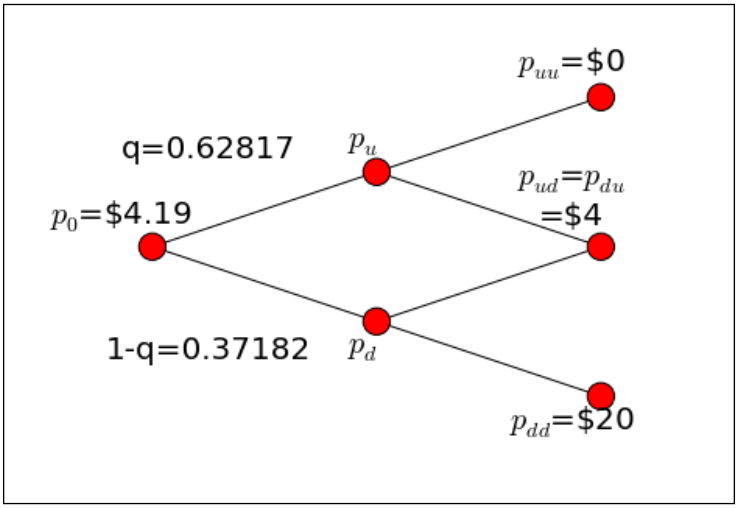
Unlike investing in stocks, investors do not have to make an upfront payment to take a position in a futures contract. In a risk-neutral sense, the expected growth rate from holding a futures contract is zero and the payoff can be written as follows:

$$1 = qu + (1-q)d$$

The risk-neutral probability q of investing in futures can be rewritten as follows:

$$q = \frac{1-d}{u-d}$$

The risk-neutral probability q of the stock given in the preceding example is calculated as 0.62817, and the payoff of the put option is given as follows:



Writing the StockOption class

Before going further in implementing the various pricing models that we are about to discuss, let's create a `StockOption` class to store and calculate the common attributes of the stock option that will be reused throughout this chapter. You can save the following code to a file named `StockOption.py`:

```
""" Store common attributes of a stock option """
import math


class StockOption(object):

    def __init__(self, S0, K, r, T, N, params):
        self.S0 = S0
        self.K = K
        self.r = r
        self.T = T
        self.N = max(1, N) # Ensure N have at least 1 time step
        self.STs = None  # Declare the stock prices tree

        """ Optional parameters used by derived classes """
        self.pu = params.get("pu", 0)  # Probability of up state
        self.pd = params.get("pd", 0)  # Probability of down state
```

```
self.div = params.get("div", 0)  # Dividend yield
self.sigma = params.get("sigma", 0)  # Volatility
self.is_call = params.get("is_call", True)  # Call or put
self.is_european = params.get("is_eu", True)  # Eu or Am

""" Computed values """
self.dt = T/float(N)  # Single time step, in years
self.df = math.exp(
    -(r-self.div) * self.dt)  # Discount factor
```

The current underlying price, strike price, risk-free rate, time to maturity, and number of time steps are compulsory common attributes for pricing options. The `params` variable is a dictionary object that accepts the required additional information pertaining to the model being used. From all of this information, the delta of the time step `dt` and the discount factor `df` are computed and can be reused throughout the pricing implementation.

Writing the BinomialEuropeanOption class

The Python implementation of the binomial option pricing model of an European option is given as the `BinomialEuropeanOption` class, which inherits the common attributes of the option from the `StockOption` class.

The `price` method of the `BinomialEuropeanOption` class is a public method that is the entry point for all the instances of this class. It calls the `_setup_parameters_` method to set up the required model parameters, and then calls the `_initialize_stock_price_tree_` method to simulate the expected values of the stock prices for the period up till *T*.

Finally, the `__begin_tree_traversal__` private method is called to initialize the payoff array and store the discounted payoff values, as it traverses the binomial tree back to the present time. The payoff tree nodes are returned as a NumPy array object, where the present value of the European option is found at the initial node.

Method names starting with double underlines "__" are private methods and can only be accessed within the same class. Method names starting with a single underline "_" are a protected method and may be overwritten by child classes. Method names not starting with an underline are public functions and may be accessed from any object.

Save this code to a file named `BinomialEuropeanOption.py`:

```
""" Price a European option by the binomial tree model """
from StockOption import StockOption
```

```
import math
import numpy as np


class BinomialEuropeanOption(StockOption):

    def __setup_parameters__(self):
        """ Required calculations for the model """
        self.M = self.N + 1  # Number of terminal nodes of tree
        self.u = 1 + self.pu  # Expected value in the up state
        self.d = 1 - self.pd  # Expected value in the down state
        self.qu = (math.exp((self.r-self.div)*self.dt) -
                   self.d) / (self.u-self.d)
        self.qd = 1-self.qu

    def _initialize_stock_price_tree_(self):
        # Initialize terminal price nodes to zeros
        self.STs = np.zeros(self.M)

        # Calculate expected stock prices for each node
        for i in range(self.M):
            self.STs[i] = self.S0*(self.u**(self.N-i))*(self.d**i)

    def _initialize_payoffs_tree_(self):
        # Get payoffs when the option expires at terminal nodes
        payoffs = np.maximum(
            0, (self.STs-self.K) if self.is_call
            else(self.K-self.STs))

        return payoffs

    def _traverse_tree_(self, payoffs):
        # Starting from the time the option expires, traverse
        # backwards and calculate discounted payoffs at each node
        for i in range(self.N):
            payoffs = (payoffs[:-1] * self.qu +
                       payoffs[1:] * self.qd) * self.df

        return payoffs

    def __begin_tree_traversal__(self):
        payoffs = self._initialize_payoffs_tree_()
        return self._traverse_tree_(payoffs)
```

```
        def price(self):
            """ The pricing implementation """
            self.__setup_parameters__()
            self._initialize_stock_price_tree_()
            payoffs = self.__begin_tree_traversal__()

         return payoffs[0]  # Option value converges to first node
```

Let's take the values from the two-step binomial tree example discussed earlier to price the European put option:

```
>>> from StockOption import StockOption
>>> from BinomialEuropeanOption import BinomialEuropeanOption
>>> eu_option = BinomialEuropeanOption
...     50, 50, 0.05, 0.5, 2,
...     {"pu": 0.2, "pd": 0.2, "is_call": False})
>>> print option.price()
4.82565175126
```

Using the binomial option pricing model gives us a present value of $4.826 for the European put option.

Pricing American options with the BinomialTreeOption class

Unlike European options that can only be exercised at maturity, American options can be exercised at any time during their lifetime.

To implement the pricing of American options in Python, we do the same with the `BinomialEuropeanOption` class and create a class named `BinomialTreeOption`. The parameters used in the `_setup_parameters_` method remain the same with the removal of an unused `M` parameter. The various methods used in American options are as follows:

- `_initialize_stock_price_tree_`: This method uses a two-dimensional NumPy array to store the expected returns of the stock prices for all time steps. This information is used to calculate the payoff values from exercising the option at each period.
- `_initialize_payoffs_tree_`: This method creates the payoff tree as a two-dimensional NumPy array, starting with the intrinsic values of the option at maturity.

- `__check_early_exercise__`: This method is a private method that returns the maximum payoff values between exercising the American option early and not exercising the option at all.
- `_traverse_tree_`: This method now includes the invocation of the `__check_early_exercise__` method to check whether it is optimal to exercise an American option early at every time step.

Implementation of the `__begin_tree_traversal__` and the `price` methods remains the same.

The `BinomialTreeOption` class can now price both European and American options when the `is_eu` key of the `params` dictionary object is set to `true` or `false` respectively, when creating an instance of the class. Save the file as `BinomialAmericanOption.py` with the following code:

```
""" Price a European or American option by the binomial tree """
from StockOption import StockOption
import math
import numpy as np


class BinomialTreeOption(StockOption):

    def _setup_parameters_(self):
        self.u = 1 + self.pu  # Expected value in the up state
        self.d = 1 - self.pd  # Expected value in the down state
        self.qu = (math.exp((self.r-self.div)*self.dt) -
                   self.d)/(self.u-self.d)
        self.qd = 1-self.qu

    def _initialize_stock_price_tree_(self):
        # Initialize a 2D tree at T=0
        self.STs = [np.array([self.S0])]

        # Simulate the possible stock prices path
        for i in range(self.N):
            prev_branches = self.STs[-1]
            st = np.concatenate((prev_branches*self.u,
                                 [prev_branches[-1]*self.d]))
            self.STs.append(st)  # Add nodes at each time step

    def _initialize_payoffs_tree_(self):
        # The payoffs when option expires
        return np.maximum(
```

```
            0, (self.STs[self.N]-self.K) if self.is_call
            else (self.K-self.STs[self.N]))

    def __check_early_exercise__(self, payoffs, node):
        early_ex_payoff = \
            (self.STs[node] - self.K) if self.is_call \
            else (self.K - self.STs[node])

        return np.maximum(payoffs, early_ex_payoff)

    def _traverse_tree_(self, payoffs):
        for i in reversed(range(self.N)):
            # The payoffs from NOT exercising the option
            payoffs = (payoffs[:-1] * self.qu +
                       payoffs[1:] * self.qd) * self.df

            # Payoffs from exercising, for American options
            if not self.is_european:
                payoffs = self.__check_early_exercise__(payoffs,
                                                        i)
        return payoffs

    def __begin_tree_traversal__(self):
        payoffs = self._initialize_payoffs_tree_()
        return self._traverse_tree_(payoffs)

    def price(self):
        self._setup_parameters_()
        self._initialize_stock_price_tree_()
        payoffs = self.__begin_tree_traversal__()

        return payoffs[0]
```

Taking the same variables in the European put option pricing example, we can create an instance of the `BinomialTreeOption` class and price this American option:

```
>>> from BinomialAmericanOption import BinomialTreeOption
>>> am_option = BinomialTreeOption(
...     50, 50, 0.05, 0.5, 2,
...     {"pu": 0.2, "pd": 0.2, "is_call": False, "is_eu": False})
>>> print am_option.price()
5.11306008282
```

The American put option is priced at $5.113. Since American options can be exercised at any time and European options can only be exercised at maturity, this added flexibility of American options increases their value over European options in certain circumstances.

For American call options on an underlying asset that does not pay dividends, there might not be an extra value over its European call option counterpart. Because of the time value of money, it costs more to exercise the American call option today before the expiration at the strike price than at a future time with the same strike price. For an in-the-money American call option, exercising the option early loses the benefit of protection against adverse price movement below the strike price as well as its intrinsic time value. With no entitlement of dividend payments, there are no incentives to exercise American call options early.

The Cox-Ross-Rubinstein model

In the preceding examples, we assumed that the underlying stock price would increase by 20 percent and decrease by 20 percent in its respective up state `u` and down state `d`. The **Cox-Ross-Rubinstein** (**CRR**) model proposes that, over a short period of time in the risk-neutral world, the binomial model matches the mean and variance of the underlying stock. The volatility of the underlying stock, or the standard deviation of returns of the stock, is taken into account as follows:

$$u = e^{\sigma\sqrt{\Delta t}}$$

$$d = \frac{1}{u} = e^{-\sigma\sqrt{\Delta t}}$$

Writing the BinomialCRROption class

The implementation of the binomial CRR model remains the same as the binomial tree discussed earlier with the exception of the model parameters `u` and `d`.

In Python, we will create a class named `BinomialCRROption` and simply inherit the `BinomialTreeOption` class. Then, all that we need to do is to override the `_setup_parameters_` method with values from the CRR model.

Instances of the `BinomialCRROption` object will invoke the price method, which will call all other methods, except the overwritten `_setup_parameters_` method, of the parent `BinomialTreeOption` class.

Save the following code to a file named `BinomialCRROption.py`:

```
""" Price an option by the binomial CRR model """
from BinomialTreeOption import BinomialTreeOption
import math


class BinomialCRROption(BinomialTreeOption):

    def _setup_parameters_(self):
        self.u = math.exp(self.sigma * math.sqrt(self.dt))
        self.d = 1./self.u
        self.qu = (math.exp((self.r-self.div)*self.dt) -
                   self.d)/(self.u-self.d)
        self.qd = 1-self.qu
```

Consider again the two-step binomial tree. The non-dividend paying stock has a current price of $50 and a volatility of 30 percent. Suppose that the risk-free rate is 5 percent per annum and the time to maturity T is 0.5 years. We would like to find the value of an European put option with a strike price K of $50 by the CRR model:

```
>>> from BinomialCRROption import BinomialTreeOption
>>> eu_option = BinomialCRROption(
...     50, 50, 0.05, 0.5, 2,
...     {"sigma": 0.3, "is_call": False})
>>> print "European put: %s" % eu_option.price()
European put: 3.1051473413
>>> am_option = BinomialCRROption(
...    50, 50, 0.05, 0.5, 2,
...    {"sigma": 0.3, "is_call": False, "is_eu": False})
>>> print "American put: %s" % am_option.price()
American put: 3.4091814964
```

Using a Leisen-Reimer tree

In the binomial models discussed earlier, we made several assumptions on the probability of up and down states as well as the resulting risk-neutral probabilities. Besides the binomial model with CRR parameters discussed, other forms of parameterization discussed widely in mathematical finance include the Jarrow-Rudd parameterization, Tian parameterization, and Leisen-Reimer parameterization. Let's take a look at the Leisen-Reimer model in detail.

Dr. Dietmar Leisen and Matthias Reimer proposed a binomial tree model with the purpose of approximating to the Black-Scholes solution as the number of steps increases. It is known as the **Leisen-Reimer** (**LR**) tree, and the nodes do not recombine at every alternate step. It uses an inversion formula to achieve better accuracy during tree transversal.

A detailed explanation of the formulas is given in the paper *Binomial Models For Option Valuation - Examining And Improving Convergence*, March 1995, which is available at `http://papers.ssrn.com/sol3/papers.cfm?abstract_id=5976`.

We will be using method two of the Peizer and Pratt Inversion function f with the following characteristic parameters:

$$f\left(z, j\left(n\right)\right) = 0.5 \mp \left[0.25 - 0.25\exp\left\{-\left(\frac{z}{n+\frac{1}{3}+\frac{0.1}{n+1}}\right)^2\left(n+\frac{1}{6}\right)\right\}\right]^{\frac{1}{2}}$$

$$j\left(n\right) = \begin{cases} n, if\ n\ is\ even \\ n+1, if\ n\ is\ odd \end{cases}$$

$$p' = f\left(d_1, j\left(n\right)\right)$$

$$p = f\left(d_2, j\left(n\right)\right)$$

$$d_1 = \frac{\log\left(\frac{S_0}{K}\right) + \left(\left(r-y\right) + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

$$d_2 = \frac{\log\left(\frac{S_0}{K}\right) - \left(\left(r-y\right) + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

$$u = e^{(r-y)\Delta t\frac{p'}{p}}$$

$$d = \frac{e^{(r-y)\Delta t} - pu}{1-p}$$

The parameter S_0 is the current stock price, K is the strike price of the option, σ is the annualized volatility of the underlying stock, T is the time to maturity of the option, r is the annualized risk-free rate, y is the dividend yield, and Δt is the time interval between each tree step.

Writing the BinomialLROption class

The Python implementation of the Leisen-Reimer tree is given in the following `BinomialLROption` class. Similar to the `BinomialCRROption` class, we can inherit the `BinomialTreeOption` class and override the variables in the `_setup_parameters_` method with those of the LR tree model:

```
""" Price an option by the Leisen-Reimer tree """
from BinomialTreeOption import BinomialTreeOption
import math


class BinomialLROption(BinomialTreeOption):

    def _setup_parameters_(self):
        odd_N = self.N if (self.N%2 == 1) else (self.N+1)
        d1 = (math.log(self.S0/self.K) +
              ((self.r-self.div) +
               (self.sigma**2)/2.) *
              self.T) / (self.sigma * math.sqrt(self.T))
        d2 = (math.log(self.S0/self.K) +
              ((self.r-self.div) -
               (self.sigma**2)/2.) *
              self.T) / (self.sigma * math.sqrt(self.T))
        pp_2_inversion = \
            lambda z, n: \
            .5 + math.copysign(1, z) * \
            math.sqrt(.25 - .25 * math.exp(
                -((z/(n+1./3.+.1/(n+1)))**2.)*(n+1./6.)))
        pbar = pp_2_inversion(d1, odd_N)

        self.p = pp_2_inversion(d2, odd_N)
        self.u = 1/self.df * pbar/self.p
        self.d = (1/self.df - self.p*self.u)/(1-self.p)
        self.qu = self.p
        self.qd = 1-self.p
```

Using the same examples as before, we can price the options using an LR tree:

```
>>> from BinomialLROption import BinomialLROption
>>> eu_option = BinomialLROption(
...     50, 50, 0.05, 0.5, 3,
...     {"sigma": 0.3, "is_call": False})
>>> print "European put: %s" % eu_option.price()
European put: 3.56742999918
>>> am_option = BinomialLROption(
...     50, 50, 0.05, 0.5, 3,
...     {"sigma": 0.3, "is_call": False, "is_eu": False})
>>> print "American put: %s" % am_option.price()
American put: 3.66817910413
```

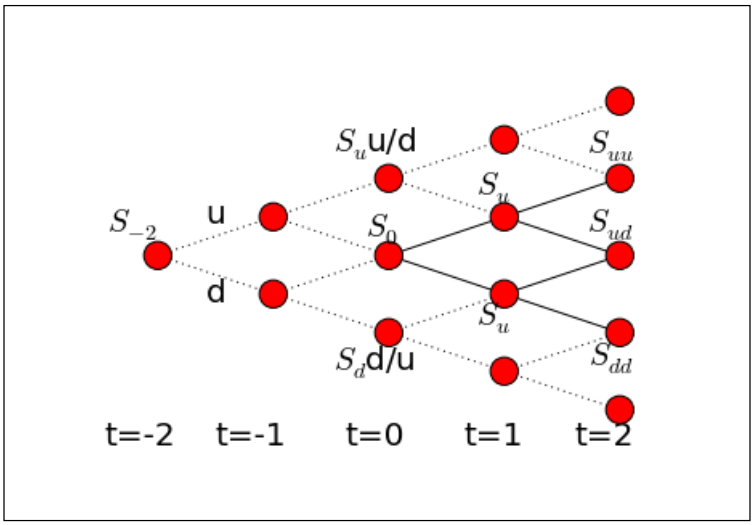
The Greeks for free

In the binomial tree pricing models that we have covered so far, we traversed up and down the tree at each point in time to determine the node values. From the information at each node, we can reuse these computed values easily. One such use is the computation of Greeks.

The Greeks measures the sensitivities of the price of derivatives such as options with respect to changes in parameters of its underlying asset, often represented by Greek letters. In mathematical finance, the common names associated with Greeks include: alpha, beta, delta, gamma, vega, theta, and rho.

Two particularly useful Greeks for options are delta and gamma. Delta measures the sensitivity of the option price with respect to the underlying asset price. Gamma measures the rate of change in delta with respect to the underlying price.

As shown in the following figure, an additional layer of nodes is added around our original two-step tree to make it a four-step tree, which extends two steps backward in time. Even with additional terminal payoff nodes, all nodes will contain the same information as our original two-step tree. Our option value of interest is now located in the middle of the tree at t=0:



Notice that at t=0 there exists two additional nodes' worth of information that we can use to compute the delta formula as follows:

$$delta = \frac{v_{up} - v_{down}}{S_0 u / d - S_0 d / u}$$

The delta formula states that the difference in the option prices in the up and down state is represented as a unit of the difference between the respective stock prices at time t=0.

Conversely, the gamma formula can be computed as follows:

$$gamma = \frac{\frac{v_{up} - v_0}{S_{0,up} - S_0} - \frac{v_0 - v_{down}}{S_0 - S_{0,down}}}{\frac{S_0 + S_{0,up}}{2} - \frac{S_0 + S_{0,down}}{2}}$$

The gamma formula states that the difference of deltas between the option prices in the up node and the down node against the initial node value are computed as a unit of the differences in price of the stock at the respective states.

Writing the BinomialLRWithGreeks class

To illustrate the computation of Greeks with the LR tree, let's create a new class named `BinomialLRWithGreeks` that inherits the `BinomialLROption` class with our own implementation of the `price` method.

In the `price` method, we will start by calling the `_setup_parameters_` method of the parent class to initialize all variables required by the LR tree. However, this time we will also call the `__new_stock_price_tree__` method, which is a new private method specially used to create an extra layer of nodes around the original tree.

The `__begin_tree_traversal__` method is called to perform the usual LR tree implementation in the parent class. The returned NumPy array object now contains the information on the three nodes at t=0, where the middle node is the option price. The payoffs in the up and down states at t=0 are in the first and last index of the array respectively.

With this information, the `price` method computes and returns the option price, and the delta and the gamma values together:

```
""" Compute option price, delta and gamma by the LR tree """
from BinomialLROption import BinomialLROption
import numpy as np


class BinomialLRWithGreeks(BinomialLROption):

    def __new_stock_price_tree__(self):
        """
        Create additional layer of nodes to our
        original stock price tree
        """
        self.STs = [np.array([self.S0*self.u/self.d,
                              self.S0,
                              self.S0*self.d/self.u])]

        for i in range(self.N):
            prev_branches = self.STs[-1]
            st = np.concatenate((prev_branches * self.u,
                                 [prev_branches[-1] * self.d]))
            self.STs.append(st)

    def price(self):
        self._setup_parameters_()
        self.__new_stock_price_tree__()
```

```
        payoffs = self.__begin_tree_traversal__()

        """ Option value is now in the middle node at t=0"""
        option_value = payoffs[len(payoffs)/2]

        payoff_up = payoffs[0]
        payoff_down = payoffs[-1]
        S_up = self.STs[0][0]
        S_down = self.STs[0][-1]
        dS_up = S_up - self.S0
        dS_down = self.S0 - S_down

        """ Get delta value """
        dS = S_up - S_down
        dV = payoff_up - payoff_down
        delta = dV/dS

        """ Get gamma value """
        gamma = ((payoff_up-option_value)/dS_up -
                 (option_value-payoff_down)/dS_down) / \
                ((self.S0+S_up)/2. - (self.S0+S_down)/2.)

        return option_value, delta, gamma
```

Using the same example from the LR tree, we can compute the option values and Greeks for an European call and put option with 300 time steps:

```
>>> from BinomialLRWithGreeks import BinomialLRWithGreeks
>>> eu_call = BinomialLRWithGreeks(
...     50, 50, 0.05, 0.5, 300, {"sigma": 0.3, "is_call": True})
>>> results = eu_call.price()
>>> print "European call values"
>>> print "Price: %s\nDelta: %s\nGamma: %s" % results
European call values
Price: 4.80386465741
Delta: 0.588801522182
Gamma: 0.0367367823884
>>> eu_put = BinomialLRWithGreeks(
...     50, 50, 0.05, 0.5, 300, {"sigma":0.3, "is_call": False})
>>> results = eu_put.price()
>>> print "European put values"
```

```
>>> print "Price: %s\nDelta: %s\nGamma: %s" % results
European put values
Price: 3.56936025883
Delta: -0.411198477818
Gamma: 0.0367367823884
```

As shown from the `price` method and results, we managed to obtain additional information on Greeks from the modified binomial tree without any extra overhead in computational complexity.

Trinomial trees in options pricing

In the binomial tree, each node leads to two other nodes in the next time step. Similarly in a trinomial tree, each node leads to three other nodes in the next time step. Besides having up and down states, the middle node of the trinomial tree indicates no change in state. When extended over more than two time steps, the trinomial tree can be thought of as a recombining tree, where the middle nodes always retain the same values as the previous time step.

Let's consider the Boyle trinomial tree, where the tree is calibrated such that the probability of up, down, and flat movements, u, d, and m with risk-neutral probabilities q_u, q_d, and q_m are as follows:

$$u = e^{\sigma\sqrt{2\Delta t}}$$

$$d = \frac{1}{u} = e^{-\sigma\sqrt{2\Delta t}}$$

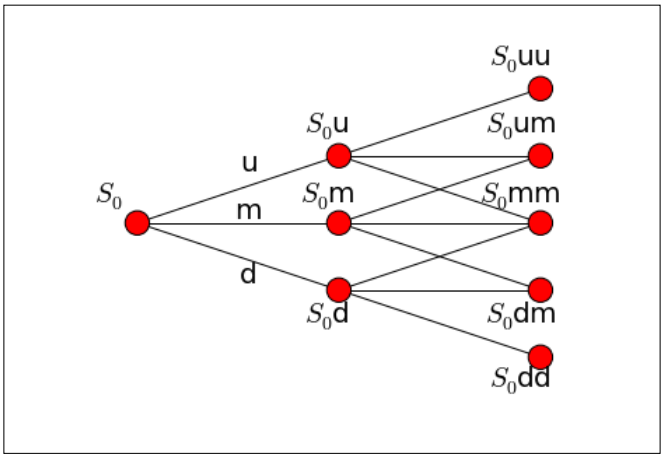
$$m = ud = 1$$

$$q_u = \left(\frac{e^{(r-v)\frac{\Delta t}{2}} - e^{\sigma}\sqrt{\frac{\Delta t}{2}}}{e^{\sigma}\sqrt{\frac{\Delta t}{2}} - e^{-\sigma}\sqrt{\frac{\Delta t}{2}}} \right)^2$$

$$q_d = \left(\frac{e^{\sigma \sqrt{\frac{\Delta t}{2}}} - e^{(r-v)\frac{\Delta t}{2}}}{e^{\sigma \sqrt{\frac{\Delta t}{2}}} - e^{-\sigma \sqrt{\frac{\Delta t}{2}}}} \right)^2$$

$$q_m = 1 - q_u - q_d$$

We can see that $ud = e^{\sigma\sqrt{2\Delta t}} e^{-\sigma\sqrt{2\Delta t}}$ recombines with $m = 1$. With calibration, the no state movement m grows at a flat rate of 1 instead of at the risk-free rate. The variable v is the annualized dividend yield and σ is the annualized volatility of the underlying stock. In general, with an increased number of nodes to process, a trinomial tree gives better accuracy than the binomial tree when fewer time steps are modeled, saving on the computation speed and resources. Refer to the following figure:



Writing the TrinomialTreeOption class

The Python implementation of the trinomial tree is given in the following `TrinomialTreeOption` class, which inherits from the `BinomialTreeOption` class.

The `_setup_parameters_` method will implement the model parameters of the trinomial tree. The `_initialize_stock_price_tree_` method will set up the trinomial tree to include the flat movement of stock prices. The `_traverse_tree_` method takes into account the middle node after discounting the payoff. Save this file as `TrinomialTreeOption.py`:

```
""" Price an option by the Boyle trinomial tree """
from BinomialTreeOption import BinomialTreeOption
import math
import numpy as np


class TrinomialTreeOption(BinomialTreeOption):

    def _setup_parameters_(self):
        """ Required calculations for the model """
        self.u = math.exp(self.sigma*math.sqrt(2.*self.dt))
        self.d = 1/self.u
        self.m = 1
        self.qu = ((math.exp((self.r-self.div) *
                             self.dt/2.) -
                    math.exp(-self.sigma *
                             math.sqrt(self.dt/2.))) /
                   (math.exp(self.sigma *
                             math.sqrt(self.dt/2.)) -
                    math.exp(-self.sigma *
                             math.sqrt(self.dt/2.))))**2
        self.qd = ((math.exp(self.sigma *
                             math.sqrt(self.dt/2.)) -
                    math.exp((self.r-self.div) *
                             self.dt/2.)) /
                   (math.exp(self.sigma *
                             math.sqrt(self.dt/2.)) -
                    math.exp(-self.sigma *
                             math.sqrt(self.dt/2.))))**2.

        self.qm = 1 - self.qu - self.qd

    def _initialize_stock_price_tree_(self):
        """ Initialize a 2D tree at t=0 """
        self.STs = [np.array([self.S0])]

        for i in range(self.N):
            prev_nodes = self.STs[-1]
```

```
            self.ST = np.concatenate(
                (prev_nodes*self.u, [prev_nodes[-1]*self.m,
                                     prev_nodes[-1]*self.d]))
            self.STs.append(self.ST)

    def _traverse_tree_(self, payoffs):
        """ Traverse the tree backwards """
        for i in reversed(range(self.N)):
            payoffs = (payoffs[:-2] * self.qu +
                       payoffs[1:-1] * self.qm +
                       payoffs[2:] * self.qd) * self.df

            if not self.is_european:
                payoffs = self.__check_early_exercise__(payoffs,
                                                        i)

        return payoffs
```

Using the same example of the binomial tree, we get the following result:

```
>>> from TrinomialTreeOption import TrinomialTreeOption
>>> eu_put = TrinomialTreeOption(
...     50, 50, 0.05, 0.5, 2,
>>> {"sigma": 0.3, "is_call": False})
>>> print "European put: %s" % eu_put.price()
European put: 3.33090549176
>>> am_option = TrinomialTreeOption(
...     50, 50, 0.05, 0.5, 2,
>>> {"sigma": 0.3, "is_call": False, "is_eu": False})
>>> print "American put: %s" % am_option.price()
American put: 3.482414539021
```

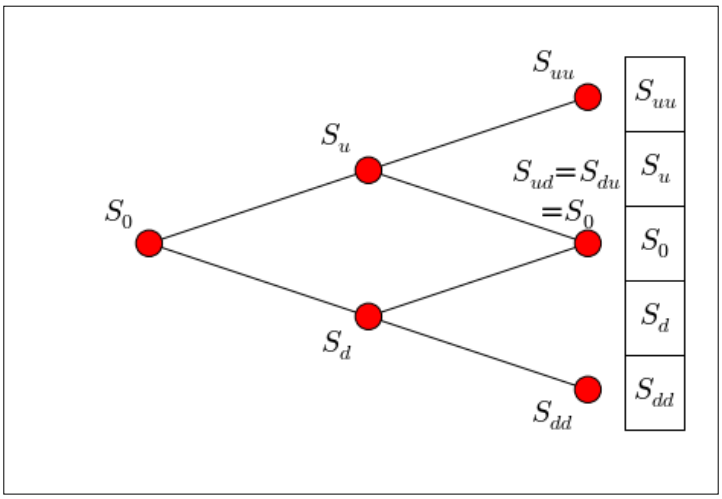
We obtain prices of $3.33 and $3.48 for the European and American put option respectively.

Lattices in options pricing

In binomial trees, each node recombines at every alternative node. In trinomial trees, each node recombines at every other node. This property of recombining trees can also be represented as lattices to help you save memory without recomputing and storing recombined nodes.

Using a binomial lattice

We will create a binomial lattice from the binomial CRR tree since, at every alternate up and down nodes, the prices recombine to the same probability of $ud = 1$. In the following figure, S_u and S_d recombine with $S_{du} = S_{ud} = S_0$. The tree can now be represented as a single list:



For a N-step binomial, a list of size $2N+1$ is required to contain the information on the underlying stock prices. For European option pricing, the odd nodes of payoffs from the list represent the option value upon maturity. The tree traverses backward to obtain the option value. For American option pricing, as the tree traverses backward, both ends of the list shrink, and the odd nodes represent the associated stock prices for any time step. Payoffs from the earlier exercise can then be taken into account.

Writing the BinomialCRROption class

Let's convert the binomial tree pricing into a lattice by CRR. We can inherit from the `BinomialCRROption` (which in turn inherits the `BinomialTreeOption` class) class and create a new class named `BinomialCRRLattice`. The following methods are overwritten with the implementation of the lattice while retaining the behavior of all the other pricing functions:

- `_setup_parameters_`: This method is overwritten to initialize the CRR parameters of the parent class as well as declaring the new variable `M` as the list size.
- `_initialize_stock_price_tree_`: This method is overwritten to set up a one-dimensional NumPy array as the lattice with the size `M`.
- `_initialize_payoffs_tree_` and `__check_early_exercise__`: These methods are overwritten to take into account the payoffs at odd nodes only.

Save this code to a file named `BinomialCRRLattice.py`:

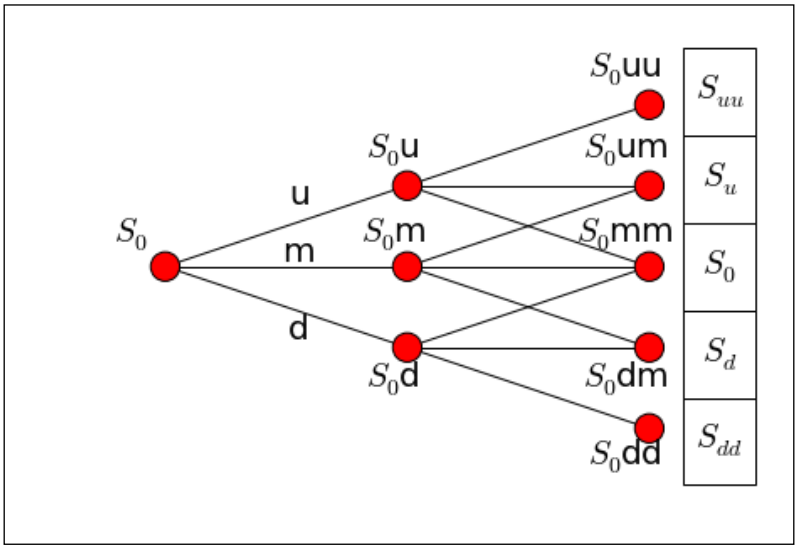
```
""" Price an option by the binomial CRR lattice """
from BinomialCRROption import BinomialCRROption
import numpy as np


class BinomialCRRLattice(BinomialCRROption):

    def _setup_parameters_(self):
        super(BinomialCRRLattice, self)._setup_parameters_()
        self.M = 2*self.N + 1

    def _initialize_stock_price_tree_(self):
        self.STs = np.zeros(self.M)
        self.STs[0] = self.S0 * self.u**self.N

        for i in range(self.M)[1:]:
            self.STs[i] = self.STs[i-1]*self.d

    def _initialize_payoffs_tree_(self):
        odd_nodes = self.STs[::2]
        return np.maximum(
            0, (odd_nodes - self.K) if self.is_call
            else(self.K - odd_nodes))

    def __check_early_exercise__(self, payoffs, node):
        self.STs = self.STs[1:-1]  # Shorten the ends of the list
        odd_STs = self.STs[::2]
        early_ex_payoffs = \
            (odd_STs-self.K) if self.is_call \
            else (self.K-odd_STs)
        payoffs = np.maximum(payoffs, early_ex_payoffs)

        return payoffs
```

Using the same stock information from our binomial CRR model example, we can price an European and American put option using the binomial lattice pricing:

```
>>> from BinomialCRRLattice import BinomialCRRLattice
>>> eu_option = BinomialCRRLattice(
...     50, 50, 0.05, 0.5, 2,
...     {"sigma": 0.3, "is_call": False})
>>> print "European put: %s" % eu_option.price()
European put: 3.1051473413
>>> am_option = BinomialCRRLattice(
...     50, 50, 0.05, 0.5, 2,
...     {"sigma": 0.3, "is_call": False, "is_eu": False})
>>> print "American put: %s" % am_option.price()
American put: 3.4091814964
```

Using the trinomial lattice

The trinomial lattice works in very much the same way as the binomial lattice. Since each node recombines at every other node instead of alternate nodes, extracting odd nodes from the list is not necessary. As the size of the list is the same as the one in the binomial lattice, there are no extra storage requirements in trinomial lattice pricing, as explained in the following figure:



Writing the TrinomialLattice class

In Python, let's create a class named `TrinomialLattice` for the trinomial lattice implementation that inherits from the `TrinomialTreeOption` class.

Just as we did for the `BinomialCRRLattice` class, the `_setup_parameters_`, `_initialize_stock_price_tree_`, `_initialize_payoffs_tree_`, and `__check_early_exercise__` methods are overwritten without having to take into account the payoffs at odd nodes:

```
""" Price an option by the trinomial lattice """
from TrinomialTreeOption import TrinomialTreeOption
import numpy as np


class TrinomialLattice(TrinomialTreeOption):

    def _setup_parameters_(self):
        super(TrinomialLattice, self)._setup_parameters_()
        self.M = 2*self.N+1

    def _initialize_stock_price_tree_(self):
        self.STs = np.zeros(self.M)
        self.STs[0] = self.S0 * self.u**self.N

        for i in range(self.M)[1:]:
            self.STs[i] = self.STs[i-1]*self.d

    def _initialize_payoffs_tree_(self):
        return np.maximum(
            0, (self.STs-self.K) if self.is_call
            else(self.K-self.STs))

    def __check_early_exercise__(self, payoffs, node):
        self.STs = self.STs[1:-1]  # Shorten the ends of the list
        early_ex_payoffs = \
            (self.STs-self.K) if self.is_call \
            else(self.K-self.STs)
        payoffs = np.maximum(payoffs, early_ex_payoffs)

        return payoffs
```

Using the same examples as before, we can price the European and American options using the trinomial lattice model:

```
>>> from TrinomialLattice import TrinomialLattice
>>> eu_option = TrinomialLattice(
...     50, 50, 0.05, 0.5, 2,
...     {"sigma": 0.3, "is_call":False})
>>> print "European put: %s" % eu_option.price()
European put: 3.33090549176
>>> am_option = TrinomialLattice(
...     50, 50, 0.05, 0.5, 2,
>>> {"sigma": 0.3, "is_call": False, "is_eu": False})
>>> print "American put: %s" % am_option.price()
American put: 3.48241453902
```

The output agrees with the results obtained from the trinomial tree option pricing model.

Finite differences in options pricing

Finite difference schemes are very much similar to trinomial tree options pricing, where each node is dependent on three other nodes with an up movement, a down movement, and a flat movement. The motivation behind the finite differencing is the application of the **Black-Scholes Partial Differential Equation** (**PDE**) framework (involving functions and their partial derivatives) whose price $S(t)$ is a function of $f(S,t)$, with r as the risk-free rate, t as the time to maturity, and σ as the volatility of the underlying security:

$$rf = \frac{df}{dt} + rS\frac{df}{dS} + \frac{1}{2}\sigma^2 S^2 \frac{d^2 f}{dt^2}$$

The finite difference technique tends to converge faster than lattices and approximates complex exotic options very well.

To solve a PDE by finite differences working backward in time, a discrete-time grid of size M by N is set up to reflect asset prices over a course of time, such that S and t take on the following values at each point on the grid:

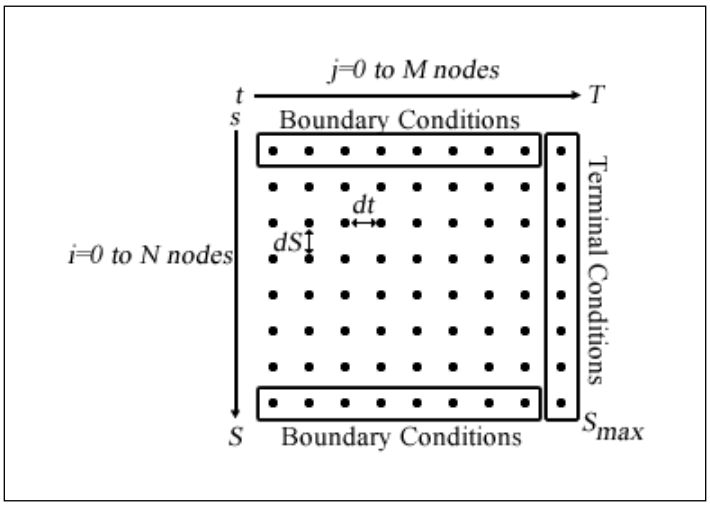
$$S = 0,\ dS,\ 2dS,\ 3dS, \ldots,\ (M-1)\ dS,\ S_{max}$$

$$t = 0,\ dt,\ 2dt,\ 3dt, \ldots,\ (N-1)\ dt,\ T$$

It follows that by grid notation, $f_{i,j} = f(i\,dS, j\,dt)$. S_{max} is a suitably large asset price that cannot be reached by the maturity time, T. dS and dt are thus intervals between each node in the grid, incremented by price and time respectively. The terminal condition at expiration time T for every value of S is $max(S-K,0)$ for a call option with strike K and $max(K-S,0)$ for a put option. The grid traverses backward from the terminal conditions, complying with the PDE while adhering to the boundary conditions of the grid, such as the payoff from an early exercise.

The boundary conditions are defined values at the extreme ends of the nodes, where i=0 and i=N for every time at *t*. Values at the boundaries are used to calculate the values of all other lattice nodes iteratively using the PDE.

A visual representation of the grid is given by the following figure. As *i* and *j* increase from the top-left corner of the grid, the price *S* tends toward `Smax` (the maximum price possible) at the bottom-right corner of the grid:



A number of ways to approximate the PDE are as follows:

- Forward difference:

$$\frac{df}{dS} = \frac{f_{i+1,j} - f_{i,j}}{dS}, \frac{df}{dt} = \frac{f_{i,j+1} - f_{i,j}}{dt}$$

- Backward difference:

$$\frac{df}{dS} = \frac{f_{i,j} - f_{i-1,j}}{dS}, \frac{df}{dt} = \frac{f_{i,j} - f_{i,j-1}}{dt}$$

- Central or symmetric difference:

$$\frac{df}{dS} = \frac{f_{i+1,j} - f_{i-1,j}}{2dS}, \frac{df}{dt} = \frac{f_{i,j+1} - f_{i,j-1}}{2dt}$$

- The second derivative:

$$\frac{d^2 f}{dS^2} = \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{dS^2}$$

Once we have the boundary conditions set up, we can now apply an iterative approach using the explicit, implicit, or Crank-Nicolson method.

The explicit method

The explicit method for approximating $f_{i,j}$ is given by:

$$rf_{i,j} = \frac{f_{i,j} - f_{i,j-1}}{dt} + ridS\frac{f_{i+1,j} - f_{i-1,j}}{2ds} + \frac{1}{2}\sigma^2 j^2 \frac{f_{i+1,j} + f_{i-1,j}}{dS^2}$$

Here, it can be seen that the first difference is the backward difference with respect to *t*, the second difference is the central difference with respect to *S*, and the third difference is the second-order difference with respect to *S*. When we rearrange the terms, we have the following equation:

$$f_{i,j} = a_i^* f_{i-1,j+1} + b_i^* f_{i,j+1} + c_i^* f_{i+1,j+1}$$

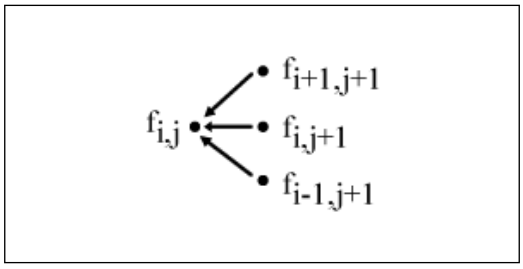
where $j = N-1, N-2, N-3, \ldots 2,1,0$ and $i = 1,2,3, \ldots M-2, M-1$:

$$a_i^* = \frac{1}{2}dt\left(\sigma^2 i^2 - ri\right)$$

$$b_i^* = 1 - dt\left(\sigma^2 i^2 - ri\right)$$

$$c_i^* = \frac{1}{2} dt\left(\sigma^2 i^2 + ri\right)$$

The iterative approach of the explicit method can be visually represented by the following figure:



Writing the FiniteDifferences class

As we will be writing the explicit, implicit, and Crank-Nicolson methods of finite differences in Python, let's write a parent class that can inherit the common properties and functions of all three methods.

We will create a class called `FiniteDifferences` that accepts and assigns all the required parameters in the `__init__` constructor method and save it as `FiniteDifferences.py`.

The `price` method is the public method used for calling the specific finite difference scheme implementation. It will invoke these methods in the following order: `_setup_boundary_conditions`, `_setup_coefficients_`, `_traverse_grid_`, and `_interpolate_`. These methods are explained as follows:

- `_setup_boundary_conditions_`: This method sets up the boundary conditions of the grid structure as a NumPy two-dimensional array
- `_setup_coefficients_`: This method sets up the necessary coefficients used for traversing the grid structure
- `_traverse_grid_`: This method iterates the grid structure backward in time, storing the calculated values toward the first column of the grid
- `_interpolate_`: Using the final calculated values on the first column of the grid, this method will interpolate these values to find the option price that closely infers the initial stock price, *S0*

All of these methods are protected methods and may be overwritten by derived classes. The `pass` keyword simply does nothing; the derived classes will provide specific implementations of these functions:

```
""" Shared attributes and functions of FD """
import numpy as np


class FiniteDifferences(object):

    def __init__(self, S0, K, r, T, sigma, Smax, M, N,
                 is_call=True):
        self.S0 = S0
        self.K = K
        self.r = r
        self.T = T
        self.sigma = sigma
        self.Smax = Smax
        self.M, self.N = int(M), int(N)  # Ensure M&N are integers
        self.is_call = is_call

        self.dS = Smax / float(self.M)
        self.dt = T / float(self.N)
        self.i_values = np.arange(self.M)
        self.j_values = np.arange(self.N)
        self.grid = np.zeros(shape=(self.M+1, self.N+1))
        self.boundary_conds = np.linspace(0, Smax, self.M+1)

    def _setup_boundary_conditions_(self):
        pass

    def _setup_coefficients_(self):
        pass

    def _traverse_grid_(self):
        """  Iterate the grid backwards in time """
        pass

    def _interpolate_(self):
        """
        Use piecewise linear interpolation on the initial
        grid column to get the closest price at S0.
        """
        return np.interp(self.S0,
```

```
                        self.boundary_conds,
                        self.grid[:, 0])
    def price(self):
        self._setup_boundary_conditions_()
        self._setup_coefficients_()
        self._traverse_grid_()
        return self._interpolate_()
```

Writing the FDExplicitEu class

The Python implementation of finite differences by the explicit method is given in the following `FDExplicitEu` class, which inherits from the `FiniteDifferences` class and overrides the required implementation methods. Save this file as `FDExplicitEu.py`:

```
""" Explicit method of Finite Differences """
import numpy as np

from FiniteDifferences import FiniteDifferences


class FDExplicitEu(FiniteDifferences):

    def _setup_boundary_conditions_(self):
        if self.is_call:
            self.grid[:, -1] = np.maximum(
                self.boundary_conds - self.K, 0)
            self.grid[-1, :-1] = (self.Smax - self.K) * \
                                 np.exp(-self.r *
                                        self.dt *
                                        (self.N-self.j_values))
        else:
            self.grid[:, -1] = \
                np.maximum(self.K-self.boundary_conds, 0)
            self.grid[0, :-1] = (self.K - self.Smax) * \
                                np.exp(-self.r *
                                       self.dt *
                                       (self.N-self.j_values))

    def _setup_coefficients_(self):
        self.a = 0.5*self.dt*((self.sigma**2) *
                              (self.i_values**2) -
                              self.r*self.i_values)
        self.b = 1 - self.dt*((self.sigma**2) *
```

```
                                (self.i_values**2) +
                                self.r)
            self.c = 0.5*self.dt*((self.sigma**2) *
                                (self.i_values**2) +
                                self.r*self.i_values)

    def _traverse_grid_(self):
        for j in reversed(self.j_values):
            for i in range(self.M)[2:]:
                self.grid[i,j] = self.a[i]*self.grid[i-1,j+1] +\
                                 self.b[i]*self.grid[i,j+1] + \
                                 self.c[i]*self.grid[i+1,j+1]
```

On completion of traversing the grid structure, the first column contains the present value of the initial asset prices at t=0. The `interp` function of NumPy is used to perform a linear interpolation to approximate the option value.

Besides using linear interpolation as the most common choice for the interpolation method, the other methods such as the spline or cubic may be used to approximate the option value.

Consider the example of an European put option. The underlying stock price is $50 with a volatility of 40 percent. The strike price of the put option is $50 with an expiration time of 5 months. The risk-free rate is 10 percent.

We can price the option using the explicit method with a `Smax` value of 100, an `M` value of 1000, and a `N` value of 100:

```
>>> from FDExplicitEu import FDExplicitEu
>>> option = FDExplicitEu(50, 50, 0.1, 5./12., 0.4, 100, 100,
...                       1000, False)
>>> print option.price()
4.07288227815
```

What happens when other values of `M` and `N` are chosen improperly?

```
>>> option = FDExplicitEu(50, 50, 0.1, 5./12., 0.4, 100, 100,
...                       100, False)
>>> print option.price()
-1.62910770723e+53
```

It appears that the explicit method of the finite difference scheme suffers from instability problems.

The implicit method

The instability problem of the explicit method can be overcome using the forward difference with respect to time. The implicit method for approximating $f_{i,j}$ is given by:

$$rf_{i,j} = \frac{f_{i,j+1} - f_{i,j}}{dt} + ridS\frac{f_{i+1,j} - f_{i-1,j}}{2dS} + \frac{1}{2}\sigma^2 j^2 \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{dS^2}$$

Here, it can be seen that the only difference between the implicit and explicit approximating scheme lies in the first difference, where the forward difference with respect to t is used in the implicit scheme. When we rearrange the terms, we have the following expression:

$$f_{i,j+1} = a_j\, f_{i-1,j} + b_i\, f_{i,j} + c_i\, f_{i+1,j}$$

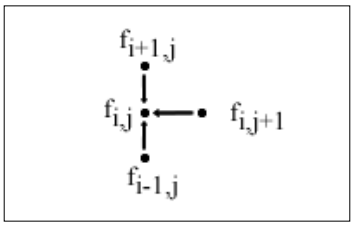
Here, $j = N-1, N-2, \ldots 2,1,0$ and $i = 1,2,3,\ldots, M-1$

$$a_i = \frac{1}{2}\left(ridt - \sigma^2 i^2 dt\right)$$

$$b_i = 1 + \sigma^2 i^2 dt + rdt$$

$$c_i = -\frac{1}{2} + \left(ridt + \sigma^2 i^2 dt\right)$$

The iterative approach of the implicit scheme can be visually represented by the following figure:



From the figure, it is intuitive to note that values of $j+1$ are required to be computed before they can be used in the next iterative step, as the grid traverses backward. In the implicit scheme, the grid can be thought of as representing a system of linear equations at each iteration, as follows:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 \\ 0 & 0 & b_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & a_{M-2} & b_{M-2} & c_{M-2} \\ 0 & 0 & 0 & 0 & a_{M-1} & b_{M-1} \end{bmatrix} \begin{bmatrix} f_{1,j} \\ f_{2,j} \\ f_{3,j} \\ \vdots \\ f_{M-2,j} \\ f_{M-1,j} \end{bmatrix} + \begin{bmatrix} a_1 f_{0,j} \\ 0 \\ 0 \\ \vdots \\ 0 \\ C_{M-1} f_{M,j} \end{bmatrix} = \begin{bmatrix} f_{1,j+1} \\ f_{2,j+1} \\ f_{3,j+1} \\ \vdots \\ f_{M-2,j+1} \\ f_{M-1,j+1} \end{bmatrix}$$

By rearranging the terms, we get the following equation:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 \\ 0 & 0 & b_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & a_{M-2} & b_{M-2} & c_{M-2} \\ 0 & 0 & 0 & 0 & a_{M-1} & b_{M-1} \end{bmatrix} \begin{bmatrix} f_{1,j} \\ f_{2,j} \\ f_{3,j} \\ \vdots \\ f_{M-2,j} \\ f_{M-1,j} \end{bmatrix} = \begin{bmatrix} f_{1,j+1} \\ f_{2,j+1} \\ f_{3,j+1} \\ \vdots \\ f_{M-2,j+1} \\ f_{M-1,j+1} \end{bmatrix} - \begin{bmatrix} a_1 f_{0,j} \\ 0 \\ 0 \\ \vdots \\ 0 \\ C_{M-1} f_{M,j} \end{bmatrix}$$

The linear system of equations can be represented in the form of $Ax = B$, where we want to solve for values of x in each iteration. Since the matrix A is tri-diagonal, we can use the LU factorization, where A=LU, for faster computation. Remember that we solved the linear system of equations using the LU decomposition in *Chapter 2*, *The Importance of Linearity in Finance*.

Writing the FDImplicitEu class

The Python implementation of the implicit scheme is given in the following `FDImplicitEu` class. We can inherit the implementation of the explicit method from the `FDExplicitEu` class discussed earlier and override the necessary methods of interest, namely the `_setup_coefficients_` and `_traverse_grid_` methods:

```
"""
Price a European option by the implicit method
of finite differences.
```

```
    """
    import numpy as np
    import scipy.linalg as linalg

    from FDExplicitEu import FDExplicitEu


    class FDImplicitEu(FDExplicitEu):

        def _setup_coefficients_(self):
            self.a = 0.5*(self.r*self.dt*self.i_values -
                          (self.sigma**2)*self.dt*(self.i_values**2))
            self.b = 1 + \
                     (self.sigma**2)*self.dt*(self.i_values**2) + \
                     self.r*self.dt
            self.c = -0.5*(self.r * self.dt*self.i_values +
                           (self.sigma**2)*self.dt*(self.i_values**2))
            self.coeffs = np.diag(self.a[2:self.M], -1) + \
                          np.diag(self.b[1:self.M]) + \
                          np.diag(self.c[1:self.M-1], 1)

        def _traverse_grid_(self):
            """ Solve using linear systems of equations """
            P, L, U = linalg.lu(self.coeffs)
            aux = np.zeros(self.M-1)

            for j in reversed(range(self.N)):
                aux[0] = np.dot(-self.a[1], self.grid[0, j])
                x1 = linalg.solve(L, self.grid[1:self.M, j+1]+aux)
                x2 = linalg.solve(U, x1)
                self.grid[1:self.M, j] = x2
```

Using the same example as with the explicit scheme, we can price an European put option using the implicit scheme:

```
>>> from FDImplicitEu import FDImplicitEu
>>> option = FDImplicitEu(50, 50, 0.1, 5./12., 0.4, 100, 100,
...                       100, False)
>>> print option.price()
4.06580193943
>>> option = FDImplicitEu(50, 50, 0.1, 5./12., 0.4, 100, 100,
...                       1000, False)
>>> print option.price()
4.07159418805
```

Given the current parameters and input data, it is observed that there are no stability issues with the implicit scheme.

The Crank-Nicolson method

Another way of avoiding the instability issue, as seen in the explicit method, is to use the Crank-Nicolson method. The Crank-Nicolson method converges much more quickly using a combination of the explicit and implicit methods, taking the average of both. This leads us to the following equation:

$$\frac{1}{2}rf_{i,j-1}+\frac{1}{2}rf_{i,j}$$
$$=\frac{f_{i,j}-f_{i,j-1}}{dt}\frac{1}{2}ridS\left(\frac{f_{i+1,j-1}-f_{i-1,j-1}}{2dS}\right)+\frac{1}{2}ridS\left(\frac{f_{i+1,j}-f_{i-1,j}}{2dS}\right)$$
$$+\frac{1}{4}\sigma^2i^2dS^2\left(\frac{f_{i+1,j-1}-2f_{i,j-1}+f_{i-1,j-1}}{dS^2}\right)$$
$$+\frac{1}{4}\sigma^2i^2dS^2\left(\frac{f_{i+1,j}-2f_{i,j}+f_{i-1,j}}{dS^2}\right)$$

This equation can also be rewritten as follows:

$$-\alpha_i f_{i-1,j-1}+\left(1-\beta_i\right)f_{i,j-1}-\gamma_i f_{i+1,j-1}=\alpha_i f_{i-1,j}+\left(1-\beta_i\right)f_{i,j-1}-\gamma_i f_{i+1,j}$$

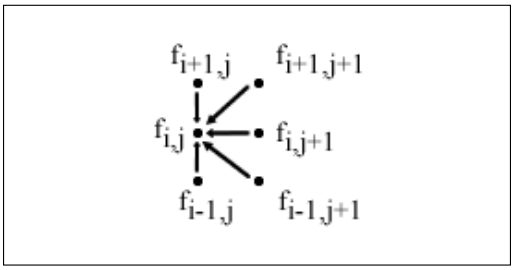
Here:

$$\alpha_i=\frac{dt}{4}\left(\sigma^2i^2-ri\right)$$

$$\beta_i=\frac{dt}{2}\left(\sigma^2i^2+ri\right)$$

$$\gamma_i=\frac{dt}{4}\left(\sigma^2i^2+ri\right)$$

The iterative approach of the implicit scheme can be visually represented by the following figure:



We can treat the equations as a system of linear equations in a matrix form:

$$M_1 f_{j-1} = M_2 f_j$$

Here:

$$M_1 = \begin{bmatrix} 1-\beta_1 & -\gamma_1 & 0 & 0 & 0 & 0 \\ -\alpha_2 & 1-\beta_2 & -\gamma_2 & 0 & 0 & 0 \\ 0 & -\alpha_3 & 1-\beta_3 & -\gamma_3 & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & -\alpha_{M-2} & 1-\beta_{M-2} & -\gamma_{M-2} \\ 0 & 0 & 0 & 0 & -\alpha_{M-1} & 1-\beta_{M-1} \end{bmatrix}$$

$$M_2 = \begin{bmatrix} 1+\beta_1 & \gamma_1 & 0 & 0 & 0 & 0 \\ \alpha_2 & 1+\beta_2 & \gamma_2 & 0 & 0 & 0 \\ 0 & \alpha_3 & 1+\beta_3 & -\gamma_3 & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & \alpha_{M-2} & 1+\beta_{M-2} & \gamma_{M-2} \\ 0 & 0 & 0 & 0 & \alpha_{M-1} & 1+\beta_{M-1} \end{bmatrix}$$

$$f_i = \left[f_{1,j}, f_{2,j}, \ldots, f_{M1-1,j} \right]^T$$

We can solve for the matrix `M` on every iterative procedure.

Writing the FDCnEu class

The Python implementation of the Crank-Nicolson method is given in the following `FDCnEu` class, which inherits from the `FDExplicitEu` class and overrides only the `_setup_coefficients_` and `_traverse_grid_` methods. Save this file as `FDCnEu.py`:

```
""" Crank-Nicolson method of Finite Differences """
import numpy as np
import scipy.linalg as linalg

from FDExplicitEu import FDExplicitEu


class FDCnEu(FDExplicitEu):

    def _setup_coefficients_(self):
        self.alpha = 0.25*self.dt*(
            (self.sigma**2)*(self.i_values**2) -
            self.r*self.i_values)
        self.beta = -self.dt*0.5*(
            (self.sigma**2)*(self.i_values**2) +
            self.r)
        self.gamma = 0.25*self.dt*(
            (self.sigma**2)*(self.i_values**2) +
            self.r*self.i_values)
        self.M1 = -np.diag(self.alpha[2:self.M], -1) + \
                  np.diag(1-self.beta[1:self.M]) - \
                  np.diag(self.gamma[1:self.M-1], 1)
        self.M2 = np.diag(self.alpha[2:self.M], -1) + \
                  np.diag(1+self.beta[1:self.M]) + \
                  np.diag(self.gamma[1:self.M-1], 1)

    def _traverse_grid_(self):
        """ Solve using linear systems of equations """
        P, L, U = linalg.lu(self.M1)

        for j in reversed(range(self.N)):
            x1 = linalg.solve(L,
                              np.dot(self.M2,
                                     self.grid[1:self.M, j+1]))
            x2 = linalg.solve(U, x1)
            self.grid[1:self.M, j] = x2
```

Using the same examples as with the explicit and implicit methods, we can price an European put option using the Crank-Nicolson method for different time point intervals:

```
>>> from FDCnEu import FDCnEu
>>> option = FDCnEu(50, 50, 0.1, 5./12., 0.4, 100, 100,
...                 100, False)
>>> print option.price()
4.072254508
>>> option = FDCnEu(50, 50, 0.1, 5./12., 0.4, 100, 100,
...                 1000, False)
>>> print option.price()
4.07223835449
```

From the observed values, the Crank-Nicolson method not only avoids the instability issue seen in the explicit scheme, but also converges faster than both the explicit and implicit methods. The implicit method requires more iterations, or bigger values of *N*, to produce values close to those of the Crank-Nicolson method.

Pricing exotic barrier options

Finite differences are especially useful in pricing exotic options pricing. The nature of the option will dictate the specifications of the boundary conditions.

In this section, we will take a look at an example of pricing a down-and-out barrier option with the Crank-Nicolson method of finite differences. Due to its relative complexity, other analytical methods, such as Monte Carlo methods, are usually employed in favor of finite difference schemes.

A down-and-out option

Let's take a look at an example of a down-and-out option. At any time during the life of the option, should the underlying asset price fall below a `Sbarrier` barrier price, the option is considered worthless. Since in the grid the finite difference scheme represents all the possible price points, we only need to consider nodes with the following price range:

$$S_{barrier} \leq S_t \leq S_{max}$$

We can then set up the boundary conditions as follows:

$$f\left(S_{max}, t\right) = 0$$

$$f\left(S_{barrier}, t\right) = 0$$

Writing the FDCnDo class

Let's create a class named `FDCnDo` that inherits from the `FDCnEu` class we discussed earlier. We can take into account the barrier price in the constructor method, while leaving the rest of the Crank-Nicolson implementation in the `FDCnEu` class unchanged:

```
"""
Price a down-and-out option by the Crank-Nicolson
method of finite differences.
"""
import numpy as np

from FDCnEu import FDCnEu


class FDCnDo(FDCnEu):

    def __init__(self, S0, K, r, T, sigma, Sbarrier, Smax, M, N,
                 is_call=True):
        super(FDCnDo, self).__init__(
            S0, K, r, T, sigma, Smax, M, N, is_call)
        self.dS = (Smax-Sbarrier)/float(self.M)
        self.boundary_conds = np.linspace(Sbarrier,
                                          Smax,
                                          self.M+1)
        self.i_values = self.boundary_conds/self.dS
```

Consider an example of a down-and-out option. The underlying stock price is $50 with a volatility of 40 percent. The strike price of the option is $50 with an expiration time of 5 months. The risk-free rate is 10 percent. The barrier price is $40.

We can price a call option and a put down-and-out option with `Smax` as 100, `M` as 120, and `N` as 500:

```
>>> from FDCnDo import FDCnDo
>>> option = FDCnDo(50, 50, 0.1, 5./12., 0.4, 40, 100, 120, 500)
>>> print option.price()
```

```
5.49156055293
>>> option = FDCnDo(50, 50, 0.1, 5./12., 0.4, 40, 100, 120, 500,
...                 False)
>>> print option.price()
0.541363502895
```

The prices of the down-and-out call and put options are $5.4916 and $0.5414 respectively.

American options pricing with finite differences

So far, we have priced European options and exotic options. Due to the probability of an early exercise nature in American options, pricing such options is less straightforward. An iterative procedure is required in the implicit Crank-Nicolson method, where the payoffs from early exercises in the current period take into account the payoffs of an early exercise in the prior period. The Gauss-Siedel iterative method is proposed in the pricing of American options in the Crank-Nicolson method.

Remember in *Chapter 2, The Importance of Linearity in Finance*, we covered the Gauss-Siedel method of solving systems of linear equations in the form of $Ax = B$. Here, the matrix A is decomposed into $A = L + U$, where L is a lower triangular matrix and U is an upper triangular matrix. Let's take a look at an example of a 4 by 4 matrix A:

$$A = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} = \begin{bmatrix} a & 0 & 0 & 0 \\ e & f & 0 & 0 \\ i & j & k & 0 \\ m & n & o & p \end{bmatrix} + \begin{bmatrix} 0 & b & c & d \\ 0 & 0 & g & h \\ 0 & 0 & 0 & l \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The solution is then obtained iteratively as follows:

$$Ax = B$$

$$\left(L + U\right)x = B$$

$$Lx = B - Ux$$

$$x_{n+1} = L^{-1}\left(B - Ux\right)$$

We can adapt the Gauss-Siedel method to our Crank-Nicolson implementation as follows:

$$r_j = M_1 f_{j-1} = M_2 f_j + \alpha_1 \begin{bmatrix} f_{0,j-1} + f_{0,j} \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

This equation satisfies the early exercise privilege equation:

$$f_{i,j-1} = \max\left(f_{i,j-1}, K - idS\right)$$

Writing the FDCnAm class

Let's create a class named `FDCnAm` that inherits from the `FDCnEu` class, which is the Crank-Nicolson method's counterpart for pricing European options. The `_setup_coefficients_` method may be reused, while overriding all other methods for the inclusion of payoffs from an early exercise, if any:

```
""" Price an American option by the Crank-Nicolson method """
import numpy as np
import sys

from FDCnEu import FDCnEu


class FDCnAm(FDCnEu):

    def __init__(self, S0, K, r, T, sigma, Smax, M, N, omega, tol,
                 is_call=True):
        super(FDCnAm, self).__init__(
            S0, K, r, T, sigma, Smax, M, N, is_call)
        self.omega = omega
        self.tol = tol
```

```
        self.i_values = np.arange(self.M+1)
        self.j_values = np.arange(self.N+1)

    def _setup_boundary_conditions_(self):
        if self.is_call:
            self.payoffs = np.maximum(
                self.boundary_conds[1:self.M]-self.K, 0)
        else:
            self.payoffs = np.maximum(
                self.K-self.boundary_conds[1:self.M], 0)

        self.past_values = self.payoffs
        self.boundary_values = self.K * \
                               np.exp(-self.r *
                                      self.dt *
                                      (self.N-self.j_values))

    def _traverse_grid_(self):
        """ Solve using linear systems of equations """
        aux = np.zeros(self.M-1)
        new_values = np.zeros(self.M-1)

        for j in reversed(range(self.N)):
            aux[0] = self.alpha[1]*(self.boundary_values[j] +
                                    self.boundary_values[j+1])
            rhs = np.dot(self.M2, self.past_values) + aux
            old_values = np.copy(self.past_values)
            error = sys.float_info.max

            while self.tol < error:
                new_values[0] = \
                    max(self.payoffs[0],
                        old_values[0] +
                        self.omega/(1-self.beta[1]) *
                        (rhs[0] -
                         (1-self.beta[1])*old_values[0] +
                         (self.gamma[1]*old_values[1])))

                for k in range(self.M-2)[1:]:
                    new_values[k] = \
                        max(self.payoffs[k],
                            old_values[k] +
                            self.omega/(1-self.beta[k+1]) *
                            (rhs[k] +
```

```
                                self.alpha[k+1]*new_values[k-1] -
                                (1-self.beta[k+1])*old_values[k] +
                                self.gamma[k+1]*old_values[k+1]))

                    new_values[-1] = \
                        max(self.payoffs[-1],
                            old_values[-1] +
                            self.omega/(1-self.beta[-2]) *
                            (rhs[-1] +
                             self.alpha[-2]*new_values[-2] -
                             (1-self.beta[-2])*old_values[-1]))

                    error = np.linalg.norm(new_values - old_values)
                    old_values = np.copy(new_values)

                self.past_values = np.copy(new_values)

            self.values = np.concatenate(([self.boundary_values[0]],
                                          new_values,
                                          [0]))

    def _interpolate_(self):
        # Use linear interpolation on final values as 1D array
        return np.interp(self.S0,
                         self.boundary_conds,
                         self.values)
```

The tolerance parameter is used in the Gauss-Siedel method as the convergence criterion. The **omega** is the over-relaxation parameter. Higher omega values give faster convergence, but this also comes with higher possibilities of the algorithm not converging.

Let's price an American call-and-put option with an underlying asset price of $50 and volatility of 40 percent, a strike price of $50, a risk-free rate of 10 percent, and an expiration date of 5 months. We choose a `Smax` value of 100, `M` as 100, `N` as 42, an omega parameter value of 1.2, and a tolerance value of 0.001:

```
>>> from FDCnDo import FDCnDo
>>> option = FDCnAm(50, 50, 0.1, 5./12., 0.4, 100, 100,
...                 42, 1.2, 0.001)
>>> print option.price()
6.10868281539
```

```
>>> option = FDCnAm(50, 50, 0.1, 5./12., 0.4, 100, 100,
...                 42, 1.2, 0.001, False)
>>> print option.price()
4.27776422938
```

The prices of the call-and-put American stock options by the Crank-Nicolson method are $6.109 and $4.2778 respectively.

Putting it all together – implied volatility modeling

In the options pricing methods we learned so far, a number of parameters are assumed to be constant: interest rates, strike prices, dividends, and volatility. Here, the parameter of interest is volatility. In quantitative research, the volatility ratio is used to forecast price trends.

To derive implied volatilities, we need to refer to *Chapter 3*, *Nonlinearity in Finance* where we discussed root-finding methods of nonlinear functions. We will use the bisection method of numerical procedures in our next example to create an implied volatility curve.

Implied volatilities of AAPL American put option

Let's consider the option data of the stock Apple (AAPL) gathered at the end of day on October 3, 2014, given in the following table. The option expires on December 20, 2014. The prices listed are the mid-points of the bid and ask prices:

Strike price	Call price	Put price
75	30	0.16
80	24.55	0.32
85	20.1	0.6
90	15.37	1.22
92.5	10.7	1.77
95	8.9	2.54
97.5	6.95	3.55
100	5.4	4.8

Strike price	Call price	Put price
105	4.1	7.75
110	2.18	11.8
115	1.05	15.96
120	0.5	20.75
125	0.26	25.81

The last traded price of AAPL was 99.62 with an interest rate of 2.48 percent and a dividend yield of 1.82 percent. The American options expire in 78 days.

Using this information, let's create a new class named `ImpliedVolatilityModel` that accepts the stock option's parameters in the `__init__` constructor method. Import the `BinomialLROption` class that we created for the Leisen-Reimer binomial tree covered in the earlier section of this chapter. Also, import the `bisection.py` file that we created using the bisection function covered in *Chapter 3*, *Nonlinearity in Finance*.

The `_option_valuation_` method accepts the strike price `K` and the volatility value `sigma` to compute the value of the option. In this example, we are using the `BinomialLROption` pricing method.

The `get_implied_volatilities` public method accepts a list of strike and option prices to compute the implied volatilities by the bisection method for every price available. Therefore, the length of the two lists must be the same.

The Python code for the `ImpliedVolatilityModel` is given as follows:

```
"""
Get implied volatilities from a Leisen-Reimer binomial
tree using the bisection method as the numerical procedure.
"""
from bisection import bisection
from BinomialLROption import BinomialLROption

class ImpliedVolatilityModel(object):

    def __init__(self, S0, r, T, div, N,
                 is_call=False):
        self.S0 = S0
        self.r = r
        self.T = T
        self.div = div
        self.N = N
        self.is_call = is_call
```

```
        def _option_valuation_(self, K, sigma):
            # Use the binomial Leisen-Reimer tree
            lr_option = BinomialLROption(
                self.S0, K, self.r,  self.T, self.N,
                {"sigma": sigma,
                 "is_call": self.is_call,
                 "div": self.div})
            return lr_option.price()

        def get_implied_volatilities(self, Ks, opt_prices):
            impvols = []
            for i in range(len(Ks)):
                # Bind f(sigma) for use by the bisection method
                f = lambda sigma: \
                    self._option_valuation_(
                        Ks[i], sigma) - opt_prices[i]
                impv = bisection(f, 0.01, 0.99, 0.0001, 100)[0]
                impvols.append(impv)
            return impvols
    if __name__ == "__main__":
```

Using this model, let's find out the implied volatilities of the American put options using this particular set of data:

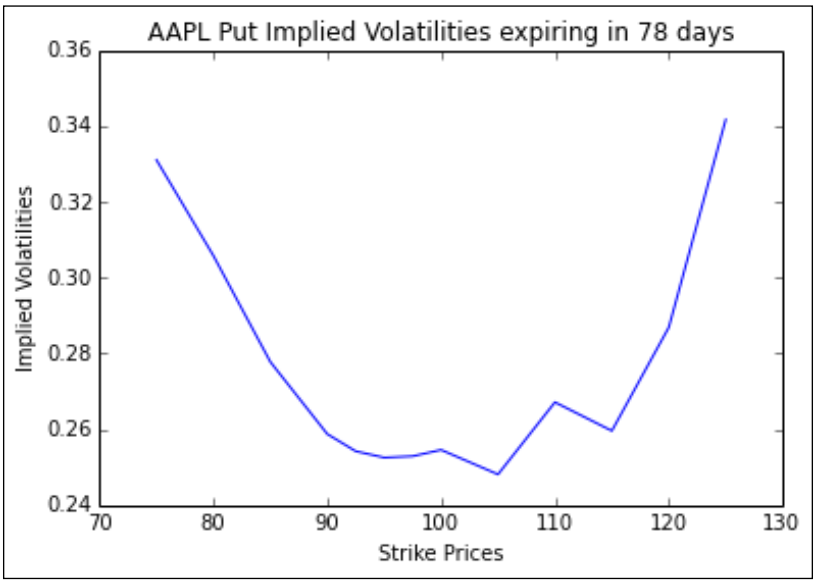
```
>>> # The data
>>> strikes = [ 75, 80, 85, 90, 92.5, 95, 97.5,
...              100, 105, 110, 115, 120, 125]
>>> put_prices = [0.16, 0.32, 0.6, 1.22, 1.77, 2.54, 3.55,
...                4.8, 7.75, 11.8, 15.96, 20.75, 25.81]
>>>
>>> model = ImpliedVolatilityModel(99.62, 0.0248, 78/365.,
...                            0.0182, 77, is_call=False)
>>> impvols_put = model.get_implied_volatilities(strikes,
                                                 put_prices)
```

The implied volatility values are now stored in the `impvols_put` variable as a list object. Let's plot these values against the strike prices so that we can get an implied volatility curve:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(strikes, impvols_put)
>>> plt.xlabel('Strike Prices')
>>> plt.ylabel('Implied Volatilities')
```

```
>>> plt.title('AAPL Put Implied Volatilities expiring in 78 days')
>>> plt.show()
```

This would give us the volatility smile, as shown in the following figure. Here, we have modeled a Leisen-Reimer tree with 77 steps, each step representing one day:



Of course, pricing an option daily may not be ideal since markets change by fractions of a millisecond. We used the bisection method to solve the implied volatility as implied by the binomial tree, as opposed to the realized volatility values directly observed from market prices.

Should we fit this curve against a polynomial curve to identify potential arbitrage opportunities? Or extrapolate the curve to derive further insights on potential opportunities from implied volatilities of far out-of-the-money and in-the-money options? Well, these questions are for options traders like you to find out!

Summary

In this chapter, we looked at a number of numerical procedures in derivative pricing the most common being options. One such procedure is the use of trees, with binomial trees being the simplest structure to model asset information, where one node extends to two other nodes in each time step, representing an up state and a down state respectively. In trinomial trees, each node extends to three other nodes in each time step, representing an up state, a down state, and a state with no movement respectively. As the tree traverses upwards, the underlying asset is computed and represented at each node. The option then takes on the structure of this tree and, starting from the terminal payoffs, the tree traverses backward and toward the root, which converges to the current discounted option price. Besides binomial and trinomial trees, trees can take on the form of the Cox-Ross-Rubinstein, Jarrow-Rudd, Tian, or Leisen-Reimer parameters.

By adding another layer of nodes around our tree, we introduced additional information from which we can derive the Greeks such as the delta and gamma without incurring additional computational cost.

Lattices were introduced as a way of saving storage costs over binomial and trinomial trees. In lattice pricing, nodes with new information are saved only once and reused later on nodes that require no change in the information.

We also discussed the finite difference schemes in option pricing, consisting of terminal and boundary conditions. From the terminal conditions, the grid traverses backward in time using the explicit method, implicit method, and the Crank-Nicolson method. Besides pricing European and American options, finite difference pricing schemes can be used to price exotic options, where we looked at an example of pricing a down-and-out barrier option.

By importing the bisection root-finding method learned in *Chapter 3*, *Nonlinearity in Finance* and the binomial Leisen-Reimer tree model in this chapter, we used market prices of an American option to create an implied volatility curve for further studies.

In the next chapter, we will take a look at working with interest rate instruments.

5
Interest Rates and Derivatives

Interest rates affect economic activities at all levels. Central banks, including the Federal Reserve (informally known as the Fed) target interest rates as a policy tool to influence economic activity. Interest rate derivatives are popular with investors who require customized cash flow needs or specific views on interest rate movements.

One of the key challenges that interest rate derivative traders face is to have a good and robust pricing procedure for these products. This involves understanding the complicated behavior of an individual interest rate movement. Several interest rate models have been proposed for financial studies. Some common models studied in finance are the Vasicek model, CIR model, and Hull-White model. These interest rate models involve modeling the short rate and rely on factors (or sources of uncertainty) with most of them using only one factor. Two-factor and multifactor interest rate models have been proposed.

In this chapter, we will cover the following topics:

- The yield curve in a normal environment and an inverted environment
- Valuing a zero-coupon bond using Python
- Bootstrapping a yield curve
- Calculating forward rates from the yield curve
- Calculating the yield to maturity and price of a bond
- Calculating the bond duration and convexity using Python
- Discussing short rate models as a function of the yield curve

- The Vasicek short rate model
- The Cox-Ingersoll-Ross short rate model
- The Rendleman and Bartter short rate model
- The Brennan and Schwartz short rate model
- Pricing a callable zero-coupon bond using finite differences
- Discussing methods of callable bond pricing

Fixed-income securities

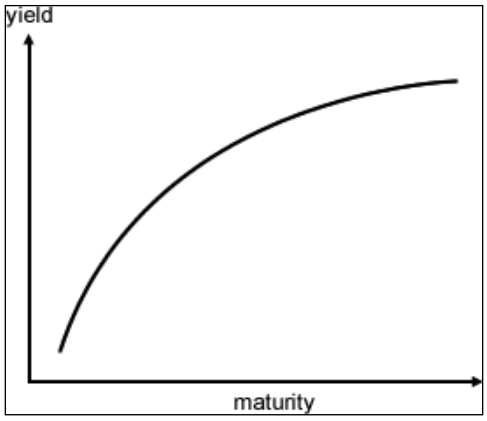
Corporations and governments issue fixed-income securities as a means of raising money. The owner of such debts lends money and expects to receive the principal when the debt matures. The issuer who wishes to borrow money may issue a fixed amount of interest payment during the lifetime of the debt at prespecified times.

The holder of debt securities, such as U.S. Treasury bills, notes, and bonds, faces the risk of default by the issuer. The federal government and municipal government are thought to face the least default risk since they can easily raise taxes and create more money to repay the outstanding debt dues.

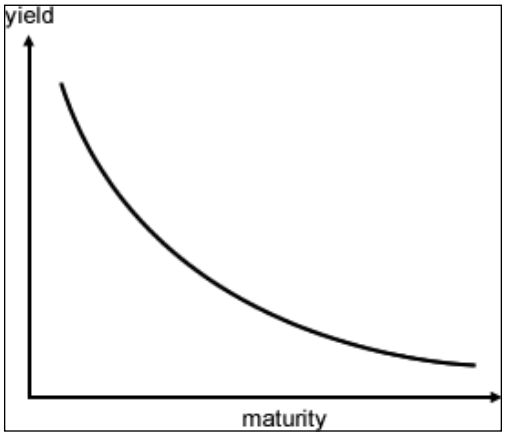
Most bonds pay a fixed amount of interest semi-annually, while some pay quarterly, or annually. These interest payments are also referred to as coupons. They are quoted as a percentage of the face value or par amount of the bond on an annual basis. For example, a five-year $10,000 Treasury bond with a coupon rate of 5 percent pays coupons of $500 in each year, or coupons of $250 every six months, up till and including the maturity date. Should the interest rates drop and new T-bonds pay a 3 percent coupon rate, the buyer of the new bond will only receive coupons of $300 annually, while existing holders of the 5 percent bond will continue to receive $500 annually. As the characteristic of the bonds influences its prices so they're closely related to current levels of the interest rates in an inverse relationship manner, the value of the bond decreases as the interest rates increase. As interest rates decrease, bond prices increase.

Yield curves

In a normal yield curve environment, long-term interest rates are higher than short-term interest rates. Investors expect to be compensated with higher returns when they lend money for a longer period since they are exposed to higher default risk. The normal or positive yield curve is said to be upward sloping, as shown in the following graph:



In certain economic conditions, the yield curve can be inverted. Long-term interest rates are lower than short-term interest rates. Such a condition occurs when the supply of money is tight. Investors are willing to forgo long-term gains to preserve their wealth in the short term. During periods of high inflation, where the inflation rate exceeds the rate of coupon interests, negative interest rates may be observed. Investors are willing to pay in the short term just to secure their long-term wealth. The inverted yield curve is said to be downward sloping, as shown in the following graph:



Valuing a zero-coupon bond

A zero-coupon bond is a bond that does not pay any periodic interest except on maturity, where the principal or face value is repaid. Zero-coupon bonds are also called pure discount bonds.

A zero-coupon bond can be valued as follows:

$$price\ of\ zero\ coupon\ bond = \frac{face\ value}{(1+y)^t}$$

Here, y is the annually compounded yield or rate of the bond, and t is the time remaining to the maturity of the bond.

Let's take a look at an example of a 5-year zero-coupon bond with a face value of $100. The yield is 5 percent, compounded annually. The price can be calculated as follows:

$$\frac{100}{(1+0.05)^5} = \$78.35$$

A simple Python zero-coupon bond calculator can be used to illustrate this example:

```
def zero_coupon_bond(par, y, t):
    """
    Price a zero coupon bond.

    Par - face value of the bond.
    y - annual yield or rate of the bond.
    t - time to maturity in years.
    """
    return par/(1+y)**t
```

Using the preceding example, we get the following result:

```
>>> print zero_coupon_bond(100, 0.05, 5)
78.3526166468
```

In the preceding example, we assumed that the investor is able to invest $78.35 at the prevailing annual interest rate of 5 percent for 5 years, compounded annually.

Spot and zero rates

As the compounding frequency increases (say, from compounded yearly to compounded daily), the future value of money reaches an exponential limit. That is to say, the present value of $100 today will reach a future value of $\$100e^{RT}$ when it is invested at a continuously compounded rate R for a period of time, T. Discounting these values for a security that pays $100 at a future time T with a continuously compounded discount rate R, its value at time zero is $\frac{100}{e^{RT}}$. This rate is known as the spot rate.

Spot rates represent the current interest rates for several maturities, should we want to borrow or lend money now. Zero rates represent the internal rate of return of zero-coupon bonds.

We can use spot rates and zero rates of bonds of different maturities to construct the present yield curve.

Bootstrapping a yield curve

Short-term spot rates can be derived directly from various short-term securities, such as zero-coupon bonds, T-bills, notes, and Eurodollar deposits. However, longer-term spot rates are typically derived from the prices of long-term bonds through a bootstrapping process, taking into account the spot rates of maturities corresponding to the coupon payment date. After obtaining short-term and long-term spot rates, the yield curve can then be constructed.

Let's illustrate the bootstrapping of the yield curve with an example. The following table shows a list of bonds with different maturities and prices:

Bond face value in Dollars	Time to maturity in years	Annual coupon in Dollars	Bond cash price in Dollars
100	0.25	0	97.50
100	0.50	0	94.90
100	1.00	0	90.00
100	1.50	8	96.00
100	2.00	12	101.60

An investor of a 3-month zero-coupon bond today at $97.50 would earn an interest of $2.50. The 3-month spot rate can be calculated as follows:

$$97.50 = \frac{100}{e^{0.25y}}$$

$$e^{0.25y} = 1.0256$$

$$y = 4ln1.0256 = 0.10127$$

Thus, the 3-month zero rate is 10.127 percent with continuous compounding. The spot rates of the zero-coupon bonds are computed in the following table:

Time to maturity in years	**Spot rate (in percent)**
0.25	10.127
0.50	10.469
1.00	10.536

Using these spot rates, we can now price the 1.5-year bond as:

$$4e^{(-0.10469)(0.5)} + 4e^{(-0.10536)(1.0)} + 104e^{(-y)(1.5)} = 96$$

To solve for y, the spot rate for the 1.5-year bond and 2-year bond is 10.681 percent and 10.808 percent respectively.

The following code is an implementation of bootstrapping a yield curve in Python. Save this code to `BootstrapYieldCurve.py`:

```
""" Bootstrapping the yield curve """
import math

class BootstrapYieldCurve():

    def __init__(self):
        self.zero_rates = dict()  # Map each T to a zero rate
        self.instruments = dict()  # Map each T to an instrument

    def add_instrument(self, par, T, coup, price,
                       compounding_freq=2):
```

```
        """  Save instrument info by maturity """
        self.instruments[T] = (par, coup, price, compounding_freq)

    def get_zero_rates(self):
        """  Calculate a list of available zero rates """
        self.__bootstrap_zero_coupons__()
        self.__get_bond_spot_rates__()
        return [self.zero_rates[T] for T in self.get_maturities()]

    def get_maturities(self):
        """ Return sorted maturities from added instruments. """
        return sorted(self.instruments.keys())

    def __bootstrap_zero_coupons__(self):
        """ Get zero rates from zero coupon bonds """
        for T in self.instruments.iterkeys():
            (par, coup, price, freq) = self.instruments[T]
            if coup == 0:
                self.zero_rates[T] = \
                    self.zero_coupon_spot_rate(par, price, T)

    def __get_bond_spot_rates__(self):
        """ Get spot rates for every marurity available """
        for T in self.get_maturities():
            instrument = self.instruments[T]
            (par, coup, price, freq) = instrument

            if coup != 0:
                self.zero_rates[T] = \
                    self.__calculate_bond_spot_rate__(
                        T, instrument)

    def __calculate_bond_spot_rate__(self, T, instrument):
        """ Get spot rate of a bond by bootstrapping """
        try:
            (par, coup, price, freq) = instrument
            periods = T * freq  # Number of coupon payments
            value = price
            per_coupon = coup / freq  # Coupon per period

            for i in range(int(periods)-1):
                t = (i+1)/float(freq)
```

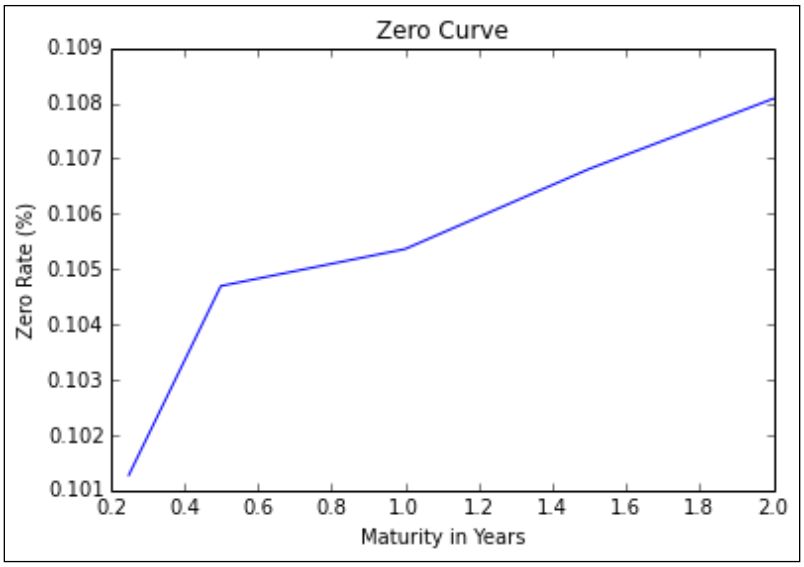
```
                spot_rate = self.zero_rates[t]
                discounted_coupon = per_coupon * \
                                    math.exp(-spot_rate*t)
                value -= discounted_coupon

            # Derive spot rate for a particular maturity
            last_period = int(periods)/float(freq)
            spot_rate = -math.log(value /
                                  (par+per_coupon))/last_period
            return spot_rate

        except:
            print "Error: spot rate not found for T=%s" % t

    def zero_coupon_spot_rate(self, par, price, T):
        """ Get zero rate of a zero coupon bond """
        spot_rate = math.log(par/price)/T
        return spot_rate
```

We can instantiate the `BootstrapYieldCurve` class, and add each bond's information from the preceding table:

```
>>> from BootstrapYieldCurve import BootstrapYieldCurve
>>> yield_curve = BootstrapYieldCurve()
>>> yield_curve.add_instrument(100, 0.25, 0., 97.5)
>>> yield_curve.add_instrument(100, 0.5, 0., 94.9)
>>> yield_curve.add_instrument(100, 1.0, 0., 90.)
>>> yield_curve.add_instrument(100, 1.5, 8, 96., 2)
>>> yield_curve.add_instrument(100, 2., 12, 101.6, 2)
>>> y = yield_curve.get_zero_rates()
>>> x = yield_curve.get_maturities()
```

Calling the `get_zero_rates` method in the class returns a list of spot rates in the same order as the maturities, which are stored in the *y* and *x* variables respectively. When we plot *x* and *y* on a graph, we get the following output:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y)
>>> plt.title("Zero Curve")
>>> plt.ylabel("Zero Rate (%)")
>>> plt.xlabel("Maturity in Years")
>>> plt.show()
```



In a normal yield curve environment, where the interest rates increase as the maturities increase, we can obtain an upward-sloping yield curve.

Forward rates

An investor who plans to invest at a later time might be curious to know what the future interest rate might look like, as implied by today's term structure of interest rates. For example, you might ask: What is the one-year spot rate one year from now? To answer this question, one can calculate forward rates for the period between T_1 and T_2 using this formula:

$$r_{forward} = \frac{r_2 T_2 - r_1 T_1}{T_2 - T_1}$$

Here, r_1 and r_2 are the continuously compounded annual interest rates at time period T_1 and T_2 respectively.

The following Python code helps us generate a list of forward rates from a list of spot rates:

```
"""
Get a list of forward rates
starting from the second time period
"""


class ForwardRates(object):

    def __init__(self):
        self.forward_rates = []
        self.spot_rates = dict()

    def add_spot_rate(self, T, spot_rate):
        self.spot_rates[T] = spot_rate

    def __calculate_forward_rate___(self, T1, T2):
        R1 = self.spot_rates[T1]
        R2 = self.spot_rates[T2]
        forward_rate = (R2*T2 - R1*T1)/(T2 - T1)
        return forward_rate

    def get_forward_rates(self):
        periods = sorted(self.spot_rates.keys())
        for T2, T1 in zip(periods, periods[1:]):
            forward_rate = \
                self.__calculate_forward_rate___(T1, T2)
            self.forward_rates.append(forward_rate)

        return self.forward_rates
```

Using spot rates derived from our preceding yield curve, we get the following result:

```
>>> fr = ForwardRates()
>>> fr.add_spot_rate(0.25, 10.127)
>>> fr.add_spot_rate(0.50, 10.469)
>>> fr.add_spot_rate(1.00, 10.536)
>>> fr.add_spot_rate(1.50, 10.681)
>>> fr.add_spot_rate(2.00, 10.808)
>>> print fr.get_forward_rates()
[10.810999999999998, 10.603, 10.971, 11.189]
```

Calling the `get_forward_rates` method of the `ForwardRates` class returns a list of forward rates, starting from the next time period.

Calculating the yield to maturity

The **yield to maturity** (**YTM**) measures the interest rate, as implied by the bond, that takes into account the present value of all the future coupon payments and the principal. It is assumed that bond holders can invest received coupons at the YTM rate until the maturity of the bond; according to risk-neutral expectations, the payments received should be the same as the price paid for the bond.

Let's take a look at an example of a 5.75 percent bond that will mature in 1.5 years with a par value of 100. The price of the bond is $95.0428 and coupons are paid semi-annually. The pricing equation can be stated as follows:

$$95.0428 = \frac{c}{\left(1+\frac{y}{n}\right)^{nT_1}} + \frac{c}{\left(1+\frac{y}{n}\right)^{nT_2}} + \frac{100+c}{\left(1+\frac{y}{n}\right)^{nT_3}}$$

Here, c is the coupon dollar amount paid at each time period, T is the time period of payment in years, n is the coupon payment frequency, and y is the YTM that we are interested to solve. To solve for YTM is typically a complex process, and most bond YTM calculators use Newton's method as an iterative process.

The bond YTM calculator is illustrated by the following Python code. Save this file as `bond_ytm.py`:

```
""" Get yield-to-maturity of a bond """
import scipy.optimize as optimize


def bond_ytm(price, par, T, coup, freq=2, guess=0.05):
    freq = float(freq)
    periods = T*freq
    coupon = coup/100.*par/freq
    dt = [(i+1)/freq for i in range(int(periods))]
    ytm_func = lambda(y): \
        sum([coupon/(1+y/freq)**(freq*t) for t in dt]) + \
        par/(1+y/freq)**(freq*t) - price

    return optimize.newton(ytm_func, guess)
```

Remember that we covered the use of Newton's method and other nonlinear function root solvers in *Chapter 3*, *Nonlinearity in Finance*. For this YTM calculator function, we used the `scipy.optimize` package to solve for the YTM.

Using the parameters from the bond example, we get the following result:

```
>>> from bond_ytm import bond_ytm
>>> ytm = bond_ytm(95.0428, 100, 1.5, 5.75, 2)
>>> print ytm
0.0936915534524
```

The YTM of the bond is 9.369 percent. Now we have a bond YTM calculator that can help us compare a bond's expected return with those of other securities.

Calculating the price of a bond

When the YTM is known, we can get back the bond price in the same way we used the pricing equation investigated earlier. Save the code as `bond_price.py`:

```
""" Get bond price from YTM """
def bond_price(par, T, ytm, coup, freq=2):
    freq = float(freq)
    periods = T*freq
    coupon = coup/100.*par/freq
    dt = [(i+1)/freq for i in range(int(periods))]
    price = sum([coupon/(1+ytm/freq)**(freq*t) for t in dt]) + \
            par/(1+ytm/freq)**(freq*T)
    return price
```

Plugging in the same values from the earlier example, we get the following result:

```
>>> from bond_price import bond_price
>>> bond_price(100, 1.5, ytm, 5.75, 2)
95.0428
```

This gives us the same original bond price discussed in the earlier example. Using the `bond_ytm` and `bond_price` functions, we can use them for further uses in bond pricing, such as finding the bond's modified duration and convexity. These two characteristics of bonds are of importance to bond traders to help them formulate various trading strategies and hedge the risk.

Bond duration

Duration is a sensitivity measure of bond prices to yield changes. Some duration measures are: effective duration, Macaulay duration, and modified duration. The type of duration that we will discuss is modified duration, which measures the percentage change in bond price with respect to a percentage change in yield (typically 1 percent or 100 basis points (bps)).

The higher the duration of a bond, the more sensitive it is to yield changes. Conversely, the lower the duration of a bond, the less sensitive it is to yield changes.

The modified duration of a bond can be thought of as the first derivative of the relationship between price and yield:

$$modified\ duration \cong \frac{P^{-} - P^{+}}{2\left(P_0\right)\left(dY\right)}$$

Here, *dy* is the given change in yield, P^{-} is the price of the bond from a decrease in yield by *dy*, P^{+} is the price of the bond from an increase in yield by *dy*, and P_0 is the initial price of the bond.

It should be noted that the duration describes the linear price-yield relationship for a small change in *Y*. Because the yield curve is not linear, using a large value of *dy* does not approximate the duration measure well.

The implementation of the modified duration calculator is given in the following Python code. The `bond_mod_duration` function uses the `bond_ytm` function as discussed earlier in this chapter to determine the yield of the bond with the given initial value. Also, it uses the `bond_price` function to determine the price of the bond with the given change in yield:

```
""" Calculate modified duration of a bond """
from bond_ytm import bond_ytm
from bond_price import bond_price


def bond_mod_duration(price, par, T, coup, freq, dy=0.01):
    ytm = bond_ytm(price, par, T, coup, freq)

    ytm_minus = ytm - dy
```

```
    price_minus = bond_price(par, T, ytm_minus, coup, freq)

    ytm_plus = ytm + dy
    price_plus = bond_price(par, T, ytm_plus, coup, freq)

    mduration = (price_minus-price_plus)/(2*price*dy)
    return mduration
```

We can find out the modified duration of the 5.75 percent bond discussed earlier that will mature in 1.5 years with a par value of 100 and a bond price of 95.0428:

```
>>> from bond_mod_duration import bond_mod_duration
>>> print bond_mod_duration(95.04, 100, 1.5, 5.75, 2, 0.01)
1.392
```

The modified duration of the bond is 1.392 years.

Bond convexity

Convexity is the sensitivity measure of the duration of a bond to yield changes. Think of convexity as the second derivative of the relationship between the price and yield:

$$convexity \cong \frac{P^{-} + P^{+} - 2P_0}{\left(P_0\right)\left(dY\right)^2}$$

Bond traders use convexity as a risk management tool to measure the amount of market risk in their portfolio. Higher convexity portfolios are less affected by interest rate volatilities than lower convexity portfolio, given the same bond duration and yield. As such, higher convexity bonds are more expensive than lower convexity ones, everything else being equal.

The implementation of a bond convexity is given as follows:

```
""" Calculate convexity of a bond """
from bond_ytm import bond_ytm
from bond_price import bond_price


def bond_convexity(price, par, T, coup, freq, dy=0.01):
```

```
    ytm = bond_ytm(price, par, T, coup, freq)

    ytm_minus = ytm - dy
    price_minus = bond_price(par, T, ytm_minus, coup, freq)

    ytm_plus = ytm + dy
    price_plus = bond_price(par, T, ytm_plus, coup, freq)

    convexity = (price_minus+price_plus-2*price)/(price*dy**2)
    return convexity
```

We can now find the convexity of the 5.75 percent bond discussed earlier that will mature in 1.5 years with a par value of 100 and a bond price of 95.0428:

```
>>> from bond_convexity import bond_convexity
>>> print bond_convexity(95.0428, 100, 1.5, 5.75, 2)
2.63395939033
```

The convexity of the bond is 2.63. For two bonds with the same par value, coupon, and maturity, their convexity may be different, depending on its location on the yield curve. Higher convexity bonds will exhibit higher price changes for the same change in yield.

Short-rate modeling

In short-rate modeling, the short rate `r(t)` is the spot rate at a particular time. It is described as a continuously compounded, annualized interest rate term for an infinitesimally short period of time on the yield curve. The short rate takes on the form of a stochastic variable in interest rate models, where the interest rates may change by small amounts at every point of time. Short rate models attempt to model the evolution of interest rates over time, and hopefully describe the economic conditions at certain periods.

Short-rate models are frequently used in the evaluation of interest rate derivatives. Bonds, credit instruments, mortgages, and loan products are sensitive to interest rate changes. Short-rate models are used as interest rate components in conjunction with pricing implementations, such as numerical methods, to help price such derivatives.

Interest rate modeling is considered a fairly complex topic since interest rates are affected by a multitude of factors, such as economic states, political decisions, government intervention, and laws of demand and supply. A number of interest rate models have been proposed to account for various characteristics of interest rates.

In this section, we will take a look at some of the most commonly used one-factor short rate models used in financial studies, namely, the Vasicek model, Cox-Ingersoll-Ross model, Rendleman and Bartter model, and Brennan and Schwartz model. Using Python, we will perform a one-path simulation to obtain a general overview of the interest rate path process. Other models commonly discussed in finance include the Ho-Lee model, Hull-White model, and Black-Karasinki model.

The Vasicek model

In the one-factor Vasicek model, the short rate is modeled as a single stochastic factor:

$$dr(t) = K\left(\theta - r(t)\right)dt + \sigma dW(t)$$

Here, K, θ, and σ are constants, and σ is the instantaneous standard deviation. $W(t)$ is the random Wiener process. The Vasicek follows an Ornstein-Uhlenbeck process, where the model reverts around the mean θ with K, the speed of mean reversion. As a result, the interest rates may become negative, which is an undesirable property in most normal economic conditions.

To help understand this model, the following Python code generates a list of interest rates:

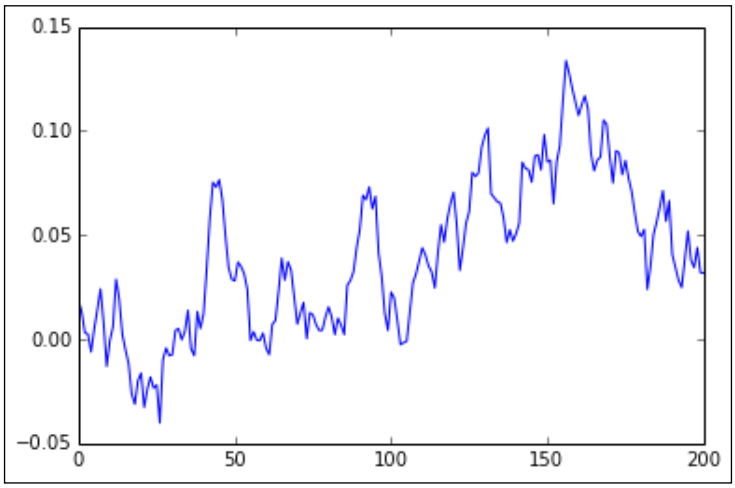
```
""" Simulate interest rate path by the Vasicek model """
import numpy as np

def vasicek(r0, K, theta, sigma, T=1., N=10, seed=777):
    np.random.seed(seed)
    dt = T/float(N)
    rates = [r0]
    for i in range(N):
        dr = K*(theta-rates[-1])*dt + sigma*np.random.normal()
        rates.append(rates[-1] + dr)
    return range(N+1), rates
```

The `vasicek` function returns a list of time periods and interest rates from the Vasicek model. It takes in a number of input parameters: `r0` is the initial rate of interest at t=0; `K`, `theta`, and `sigma` are constants; `T` is the period in terms of number of years; `N` is the number of intervals for the modeling process; and `seed` is the initialization value for NumPy's standard normal random number generator.

Assume that the current interest rate is 1.875 percent, `K` is `0.2`, `theta` is `0.01`, and `sigma` is `0.012`. We will use a `T` value of `10` and `N` value of `200` to model the interest rates as follows:

```
>>> x, y = vasicek(0.01875, 0.20, 0.01, 0.012, 10., 200)
>>>
>>> import matplotlib.pyplot as plt
>>> plt.plot(x,y)
>>> plt.show()
```

After running the above commands, we will get the following output:



In this example, we will run just one simulation to observe what the interest rates from the Vasicek model will look like. As observed, the interest rates did become negative at some point and grew on an average at 0.01.

The Cox-Ingersoll-Ross model

The **Cox-Ingersoll-Ross** (**CIR**) model is a one-factor model that was proposed to address the negative interest rates found in the Vasicek model. The process is given as:

$$dr(t) = K(\theta - r(t))dt + \sigma\sqrt{r(t)}dW(t)$$

The term $\sqrt{r(t)}$ increases the standard deviation as the short rate increases.

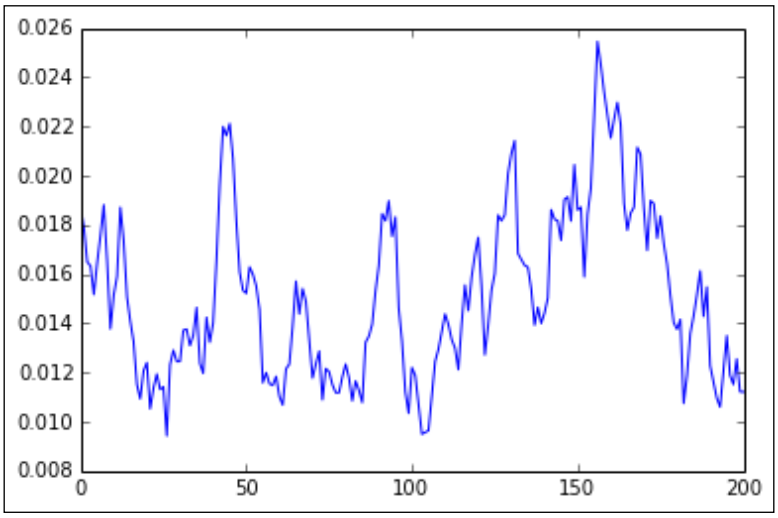
Now the `vasicek` function can be rewritten as the CIR model in Python:

```
""" Simulate interest rate path by the CIR model """
import math
import numpy as np

def cir(r0, K, theta, sigma, T=1.,N=10,seed=777):
    np.random.seed(seed)
    dt = T/float(N)
    rates = [r0]
    for i in range(N):
        dr = K*(theta-rates[-1])*dt + \
            sigma*math.sqrt(rates[-1])*np.random.normal()
        rates.append(rates[-1] + dr)
    return range(N+1), rates
```

Using the same example given in the Vasicek section, assume that the current interest rate is 1.875 percent, `K` is `0.2`, `theta` is `0.01`, and `sigma` is `0.012`. We will use a `T` value as `10` and `N` as `200` to model the interest rates as follows:

```
>>> x, y = cir(0.01875, 0.20, 0.01, 0.012, 10., 200)
>>>
>>> import matplotlib.pyplot as plt
>>> plt.plot(x,y)
>>> plt.show()
```

Here is the output for the preceding commands:



Observe that the CIR interest model does not have negative interest rate values.

The Rendleman and Bartter model

In the Rendleman and Bartter model, the short rate process is given as:

$$dr(t) = \theta r(t)\,dt + \sigma r(t)\,dW(t)$$

Here, the instantaneous drift is $\theta r(t)$ with an instantaneous standard deviation $\sigma r(t)$. The Rendleman and Bartter model can be thought of as a geometric Brownian motion, akin to a stock price stochastic process that is log-normally distributed. This model lacks the property of mean reversion. Mean reversion is a phenomenon where the interest rates seem to be pulled back toward a long-term average level.

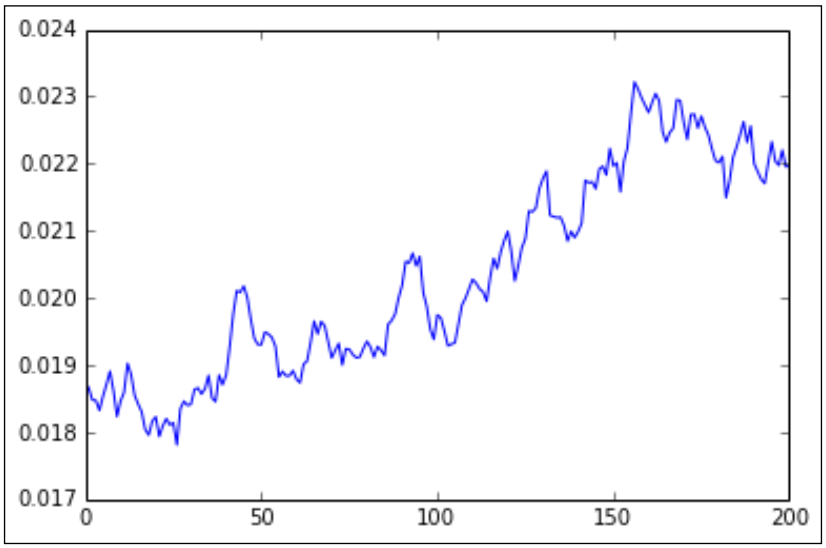
The following Python code models the Rendleman and Bartter interest rate process:

```
""" Simulate interest rate path by the Rendleman-Barter model """
import numpy as np

def rendleman_bartter(r0, theta, sigma, T=1.,N=10,seed=777):
    np.random.seed(seed)
    dt = T/float(N)
    rates = [r0]
    for i in range(N):
        dr = theta*rates[-1]*dt + \
             sigma*rates[-1]*np.random.normal()
        rates.append(rates[-1] + dr)
    return range(N+1), rates
```

We will continue to use the example from the previous sections and compare the model. Assume that the current interest rate is 1.875 percent, `theta` is `0.01`, and `sigma` is `0.012`. We will use a `T` value as `10` and `N` as `200` to model the interest rates as follows:

```
>>> x, y = rendleman_bartter(0.01875, 0.01, 0.012, 10., 200)
>>>
>>> import matplotlib.pyplot as plt
>>> plt.plot(x,y)
>>> plt.show()
```

The following graph is the output for the preceding commands:



In general, this model lacks the property of mean reversion and grows toward a long-term average level.

The Brennan and Schwartz model

The Brennan and Schwartz model is a two-factor model where the short-rate reverts toward a long rate as the mean, which also follows a stochastic process. The short-rate process is given as:

$$dr(t) = K\left(\theta - t(t)\right)dt + \sigma r(t)\,dW(t)$$

It can be seen that the Brennan and Schwartz model is another form of a geometric Brownian motion.

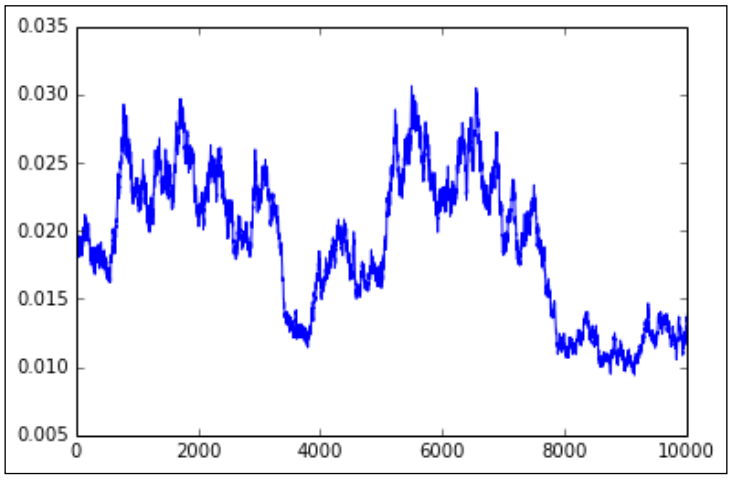
Our Python code can now be implemented as follows:

```
""" Simulate interest rate path by the Brennan Schwartz model """
import numpy as np

def brennan_schwartz(r0, K, theta, sigma, T=1., N=10, seed=777):
    np.random.seed(seed)
    dt = T/float(N)
    rates = [r0]
    for i in range(N):
        dr = K*(theta-rates[-1])*dt + \
            sigma*rates[-1]*np.random.normal()
        rates.append(rates[-1] + dr)
    return range(N+1), rates
```

Assume that the current interest rate is 1.875 percent, `K` is `0.2`, `theta` is `0.01`, and `sigma` is `0.012`. We will use a `T` value as `10` and `N` as `10000` to model the interest rates as follows:

```
>>> x, y = brennan_schwartz(0.01875, 0.20, 0.01, 0.012, 10.,
...                         10000)
>>>
>>> import matplotlib.pyplot as plt
>>> plt.plot(x,y)
>>> plt.show()
```

After running the above commands, we will get the following output:



Bond options

When bond issuers, such as corporations, issue bonds, one of the risks they face is the interest rate risk. When interest rates decrease, bond prices increase. While existing bondholders will find their bonds more valuable, bond issuers, on the other hand, find themselves in a losing position since they will be issuing higher interest payments than the prevailing interest rate. Conversely, when interest rates increase, bond issuers are at an advantage since they are able to continue issuing the same low interest payments as agreed on the bond contract specifications.

To capitalize on interest rate changes, bond issuers may embed options within a bond. This allows the issuer the right, but not the obligation, to buy or sell the issued bond at a predetermined price during a specified period of time. An American type of bond option allows the issuer to exercise the rights of the option at any point of time during the lifetime of a bond. An European type of bond option allows the issuer to exercise the rights of the option at a specific date. The exact style of the date of exercise varies from bond option to bond option. Some issuers may choose to exercise the right of the bond option when the bond has been in circulation in the market for over a year. Some issuers may choose to exercise the bond option at one of several specific dates. Regardless of the exercise dates of the bond, you may price the bond with an embedded option as:

$$price\ of\ bond = price\ of\ bond\ with\ no\ option - price\ of\ embedded\ option$$

The pricing of a bond with no option is fairly straightforward: the present value of the bond to be received at a future date, including all coupon payments. A number of assumptions are to be made on the theoretical interest rates into the future at which the coupon payments may be reinvested. One such assumption might be the movement of interest rates as implied by short rate models, which we have covered in the preceding section. Another assumption might be the movement of interest rates within a binomial or trinomial tree. For simplicity, in bond pricing studies, we will price zero-coupon bonds that will not issue coupons during the lifetime of the bond.

To price an option, you would have to determine available exercise dates. Starting from the future value of the bond, the bond price is compared against the exercise price of the option and traverses back to the present time using a numerical procedure, such as a binomial tree. This price comparison is performed at time points, where the bond option may be exercised. By the no-arbitrage theory, accounting for the present excess values of the bond when exercised, we obtain the price of the option. For simplicity, in bond pricing studies in the later sections of this chapter, we will treat the embedded option of zero-coupon bonds as an American option.

Callable bonds

In an economic condition where there are high interest rates, bond issuers are likely at risk of facing an interest rate decrease and having to continue with issuing higher interest payments than the prevailing interest rate. As such, they may choose to issue callable bonds. The callable bond contains an embedded agreement to redeem the bond at agreed dates. Existing bond holders are considered to have sold a call option to the bond issuer. In the event that the interest rates do fall and the corporation has the rights to exercise the option to buy back the bond during that period at a specific price, they may choose to do so. The company can then issue new bonds at lower interest rates. This also means that the company is able to raise more capital in the form of higher bond prices.

Puttable bonds

Unlike callable bonds, the owner of puttable bonds has the right, but not the obligation, to sell the bond back to the issuer at an agreed price during a certain period. Owners of puttable bonds are considered to have bought a put option from the bond issuer. When interest rates increase, values of existing bonds become less valuable and puttable bond holders are more incentivized to exercise the right to sell the bond at a higher exercise price. Since puttable bonds are more beneficial to buyers than to the issuers, they are generally less common than callable bonds. Variants of puttable bonds can be found in the form of loan and deposition instruments. A customer who has placed a fixed-rate deposit with a financial institution receives interest payments on specified dates. They are entitled to withdraw the deposit at any time. As such, a fixed-rate deposit instrument can be thought of as a bond with an embedded American put option.

An investor who wishes to borrow money from a bank enters a loan agreement, making interest payments during the lifetime of the agreement until the debt, together with the principal amount and agreed interests, is fully repaid. The bank can be considered as buying a put option on a bond. Under certain circumstances, the bank may exercise the right to redeem the full value of the loan agreement.

Thus, the price of puttable bonds can be thought of as:

$$price\ of\ \ putable\ bond = price\ of\ \ bond\ with\ no\ option + price\ of\ \ put\ option$$

Convertible bonds

Convertible bonds are issued by companies and contain an embedded option that allows the holder to convert the bond into a number of shares of common stock. The amount of shares to be converted for a bond is defined as the conversion ratio, which is determined such that the dollar amount of shares is the same as the value of the bond.

Convertible bonds have similarities with callable bonds. They allow the bond holders to exercise the bond for an equivalent amount of shares at the specified conversion ratio at agreed times. Convertible bonds typically issue lower coupon rates than nonconvertible bonds, to compensate for the additional value of the right to exercise.

When convertible bond holders exercise their rights into stocks, the company's debts are reduced. On the other hand, the company's stocks become more diluted as the number of shares in the circulation increases, and the company's stock price is expected to fall.

As the company's stock price increases, convertible bond prices tend to increase. Conversely, as the company's stock price decreases, convertible bond prices tend to decrease.

Preferred stocks

Preferred stocks are stocks that have bond-like qualities. Owners of preferred stocks have seniority of claim of dividend payments over common stocks, which are usually negotiated as a fixed percentage of their par value. Although there is no guarantee of dividend payments, all dividends are paid on preferred stock first over common stock. In certain agreements on preferred stocks, dividends that are not paid as agreed may accumulate until they are paid at a later time. These preferred stocks are known as **cumulative**.

Prices of preferred stocks typically move in tandem with their common stock. They may have voting rights associated with common shareholders. In the event of bankruptcy, preferred stocks have a first lien of its par value upon liquidation.

Pricing a callable bond option

In this section, we will take a look at pricing a callable bond. We assume that the bond to be priced is a zero-coupon paying bond with an embedded European call option. The price of a callable bond can be thought of as:

$$price\ of\ callable\ bond = price\ of\ bond\ with\ no\ option - price\ of\ call\ option$$

Pricing a zero-coupon bond by the Vasicek model

The value of a zero-coupon bond with a par value of 1 at time t and prevailing interest rate r is defined as:

$$P(t) = e^{-rdt}$$

Since the interest rate r is always changing, we will rewrite the zero-coupon bond as:

$$P(t) = e^{-\int_t^T r(s)ds}$$

Now, the interest rate *r* is a stochastic process that accounts for the price of the bond from time `t` to `T`, where `T` is the time to maturity of the zero-coupon bond.

To model the interest rate *r* we can use one of the short rate models as discussed in this chapter as a stochastic process. For this purpose, we will use the Vasicek model to model the short rate process.

The expectation of a log-normally distributed variable X is given by:

$$X = e^{u}$$

$$E\left[X\right] = E\left[e^{u}\right] = e^{u+\frac{\sigma^2}{2}}$$

Taking moments of the log-normally distributed variable *X*:

$$E\left[e^{su}\right] = e^{su+\frac{s^2\sigma^2}{2}}$$

We obtained the expected value of a log-normally distributed variable, which we will use in the interest rate process for the zero-coupon bond.

Remember the Vasicek short-rate process model:

$$dr(t) = K\left(\theta - r(t)\right)dt + \sigma dW(t)$$

Then, *r(t)* is derived as:

$$r(t) = \theta + \left(r_0 - \theta\right)e^{-kt} + \sigma e^{-kt}\int_0^t e^{ks}dB$$

Using the characteristic equation and the interest rate movements of the Vasicek model, we can rewrite the zero-coupon bond price in terms of expectations:

$$P(t) = E\left[e^{-\int_t^t r(s)ds}\right]$$

$$P(\tau) = A(\tau)e^{-r_t B(\tau)}$$

Here:

$$A(\tau) = e^{\left(\theta - \frac{\sigma^2}{2k^2}\right)(B(\tau)-\tau) - \frac{\sigma^2}{4k}B(\tau)^2}$$

$$B(\tau) = \frac{1 - e^{-k\tau}}{k}$$

$$\tau = T - t$$

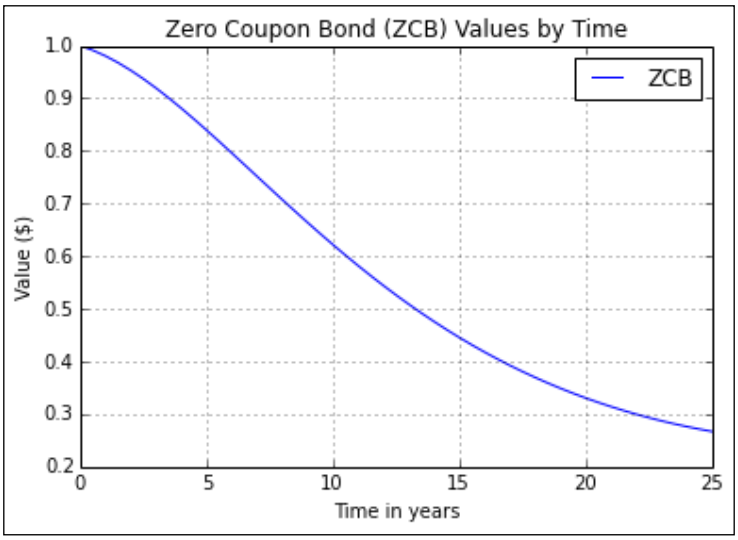
The Python implementation of the zero-coupon bond price is given in the `ExactZCB` function:

```
import numpy as np

""" Get zero coupon bond price by Vasicek model """
def exact_zcb(theta, kappa, sigma, tau, r0=0.):
    B = (1 - np.exp(-kappa*tau)) / kappa
    A = np.exp((theta-(sigma**2)/(2*(kappa**2))) *
               (B-tau) - (sigma**2)/(4*kappa)*(B**2))
    return A * np.exp(-r0*B)
```

For example, we are interested in finding out the prices of zero-coupon bond prices for a number of maturities. We model the Vasicek short-rate process with a `theta` value of `0.5`, `kappa` value of `0.02`, `sigma` value of `0.02`, and an initial interest rate `r0` of `0.015`. Plugging these values into the `ExactZCB` function, we obtain zero-coupon bond prices, for the time period from 0 to 25 years with intervals of 0.5 years, and plot out the graph:

```
>>> Ts = np.r_[0.0:25.5:0.5]
>>> zcbs = [exact_zcb(0.5, 0.02, 0.03, t, 0.015) for t in Ts]
>>>
>>> import matplotlib.pyplot as plt
>>> plt.title("Zero Coupon Bond (ZCB) Values by Time")
>>> plt.plot(Ts, zcbs, label='ZCB')
>>> plt.ylabel("Value ($)")
>>> plt.xlabel("Time in years")
>>> plt.legend()
>>> plt.grid(True)
>>> plt.show()
```

The following graph is the output for the preceding commands:



Value of early-exercise

Issuers of callable bonds may redeem the bond at an agreed price as specified in the contract. To price such a bond, the discounted early-exercise values can be defined as:

$$discounted\ early\ exercise\ value = ke^{-rt}$$

Here, k is the price ratio of the strike price to the par value and r is the interest rate for the strike price.

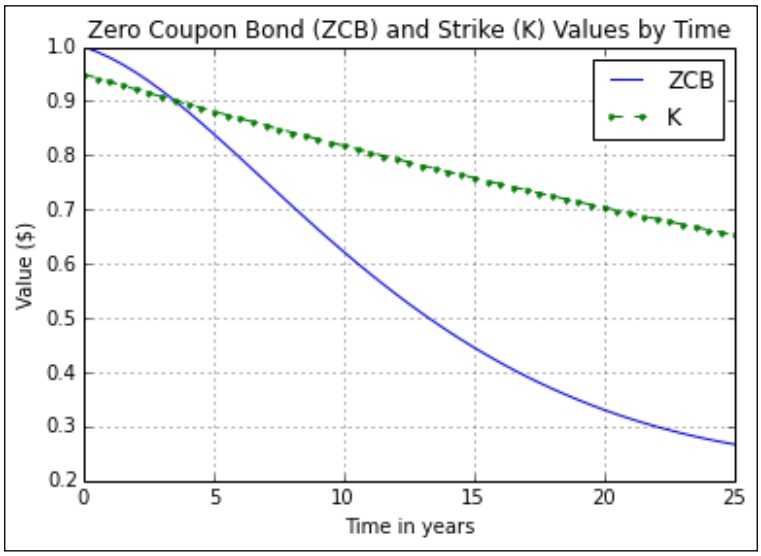
The Python implementation of the early-exercise option can then be written as follows:

```
import math
def exercise_value(K, R, t):
    return K*math.exp(-R*t)
```

In the preceding example, we are interested in valuing a call option with a strike ratio of `0.95` and an initial interest rate of 1.5 percent. We can then plot the values as a function of time and superimpose them onto a graph of zero-coupon bond prices to give us a better visual representation of the relationship between zero-coupon bond prices and callable bond prices:

```
>>> Ts = np.r_[0.0:25.5:0.5]
>>> Ks = [exercise_value(0.95, 0.015, t) for t in Ts]

>>> zcbs = [exact_zcb(0.5, 0.02, 0.03, t, 0.015) for t in Ts]
>>> import matplotlib.pyplot as plt
>>> plt.title("Zero Coupon Bond (ZCB) "
...           "and Strike (K) Values by Time")
>>> plt.plot(Ts, zcbs, label='ZCB')
>>> plt.plot(Ts, Ks, label='K', linestyle="--", marker=".")
>>> plt.ylabel("Value ($)")
>>> plt.xlabel("Time in years")
>>> plt.legend()
>>> plt.grid(True)
>>> plt.show()
```

Here is the output for the preceding commands:



From the preceding graph, we can approximate the price of callable zero-coupon bond prices. Since the bond issuer owns the call, the price of the callable zero-coupon bond can be stated as:

$$callable\ zero\ coupon\ bond\ price = \min\left(ZCB, K\right)$$

This callable bond price is an approximation, given the current interest rate level. The next step would be to treat early-exercise by going through a form of policy iteration, which is a cycle used to determine optimum early-exercise values and their effect on other nodes, and check whether they become due for an early exercise. In practice, such an iteration only occurs once.

Policy iteration by finite differences

So far, we have used the Vasicek model in our short rate process for modeling a zero-coupon bond. We can undergo policy iteration by finite differences to check for early-exercise conditions and their effect on other nodes. We will use the implicit method of finite differences for the numerical pricing procedure, as discussed in *Chapter 4*, *Numerical Procedures*.

Let's create a class named `VasicekCZCB` that will incorporate all the methods used for implementing the pricing of callable zero-coupon bonds by the Vasicek model. The full Python code of this class can be found at the end of this section.

The methods used are as follows:

- `vasicek_czcb_values(self, r0, R, ratio, T, sigma, kappa, theta, M, prob=1e-6, max_policy_iter=10, grid_struct_const=0.25, rs=None)`: This method is the point of entry to kick-start the pricing process. The variable `r0` is the short-rate at time $t = 0$; `R` is the strike zero rate for the bond price; `ratio` is the strike price per par value of the bond; `T` is the time to maturity; `sigma` is the volatility of the short rate r; `kappa` is the rate of mean reversion; `theta` is the mean of the short rate process; `M` is the number of steps in the finite differences scheme, `prob` is the probability on the normal distribution curve used by the `vasicek_limits` method to determine short rates; `max_policy_iter` is the maximum number of policy iterations used to find early-exercise nodes; `grid_struct_const` is the maximum threshold of `dt` movement that determines `N` in the `calculate_N` method; and `rs` is the list of interest rates from which the short rate process follows. This method returns a list of evenly spaced short rates and a list of option prices.

- `vasicek_params(self, r0, M, sigma, kappa, theta, T, prob, grid_struct_const=0.25, rs=None)`: This method computes the implicit scheme parameters for the Vasicek model. It returns comma-separated values of `r_min`, `dr`, `N`, and `dt`. If no value is supplied to `rs`, values of `r_min` to `r_max` will be automatically generated by the `vasicek_limits` method as a function of `prob` following a normal distribution.
- `vasicek_limits(self, r0, sigma, kappa, theta, T, prob=1e-6)`: This method computes the minimum and maximum of the Vasicek interest rate process by a normal distribution process. The expected value of the short rate process `r(t)` under the Vasicek model is given as:

$$E\left[r\left(t\right)\right] = \theta + \left(r_0 - \theta\right)e^{-kt}$$

  The variance is defined as:

$$Var\left[r\left(t\right)\right] = \frac{\sigma^2}{2k}\left(1 - e^{-2kt}\right)$$

  The function returns a tuple of the minimum and maximum interest rate level as defined by the probability for the normal distribution process.

- `vasicek_diagonals(self, sigma, kappa, theta, r_min, dr, N, dtau)`: This method returns the diagonals of the implicit scheme of finite differences, where:

$$sub - diagonals, a = k\left(\theta - r_i\right)\frac{dt}{2dr} - \frac{1}{2}\sigma^2\frac{dt}{dr^2}$$

$$diagonals, b = 1 + r_i dt + \sigma^2\frac{dt}{dr^2}$$

$$super - diagonals, c = k\left(\theta - r_i\right)\frac{dt}{2dr} - \frac{1}{2}\sigma^2\frac{dt}{dr^2}$$

  The boundary conditions are implemented using Neumann boundary conditions.

- `check_exercise(self, V, eex)`: This method returns a list of Boolean values, indicating the indices suggesting optimum payoff from an early exercise.
- `exercise_call_price(self, R, ratio, tau)`: This method returns the discounted value of the strike price as a ratio.
- `vasicek_policy_diagonals(self, subdiagonal, diagonal, superdiagonal, v_old, v_new, eex)`: This method is used by the policy iteration procedure that updates the sub-diagonals, diagonals, and super diagonals for one iteration. In indices, where an early exercise is carried out, the sub-diagonals and super diagonals will have these values set to 0 and the remaining values on the diagonal. The method returns comma-separated values of the new sub-diagonal, diagonal, and super-diagonal values.
- `iterate(self, subdiagonal, diagonal, superdiagonal, v_old, eex, max_policy_iter=10)`: This method performs the implicit scheme of finite differences by performing a policy iteration, where each cycle involves solving the tridiagonal systems of equations, calling the `vasicek_policy_diagonals` method to update the three diagonals, and returns the callable zero-coupon bond price if there are no further early-exercise opportunities. It also returns the number of policy iterations performed.
- `tridiagonal_solve(self, a, b, c, d)`: This method is the implementation of the Thomas algorithm for solving tridiagonal systems of equations. The systems of equations may be written as:

$$a_i x_{i-1} + b_i x_i + c_i a_i x_{i+1} = d_i$$

This equation is represented in matrix form:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & \ddots & 0 \\ 0 & \ddots & \ddots & c_{n-1} \\ 0 & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_1 \\ \vdots \\ d_n \end{bmatrix}$$

Here, a is a list for the sub-diagonals, b is a list for the diagonal, and c is the super diagonal of the matrix.

With these methods defined, we can now run our code and price a callable zero-coupon bond by the Vasicek model.

The implementation of the `VasicekCZCB` class in Python is given as follows:

```
""" Price a callable zero coupon bond by the Vasicek model """
import math
import numpy as np
import scipy.stats as st


class VasicekCZCB:

    def __init__(self):
        self.norminv = st.distributions.norm.ppf
        self.norm = st.distributions.norm.cdf

    def vasicek_czcb_values(self, r0, R, ratio, T, sigma, kappa,
                            theta, M, prob=1e-6,
max_policy_iter=10,
                            grid_struct_const=0.25, rs=None):
        r_min, dr, N, dtau = \
            self.vasicek_params(r0, M, sigma, kappa, theta,
                                T, prob, grid_struct_const, rs)
        r = np.r_[0:N]*dr + r_min
        v_mplus1 = np.ones(N)

        for i in range(1, M+1):
            K = self.exercise_call_price(R, ratio, i*dtau)
            eex = np.ones(N)*K
            subdiagonal, diagonal, superdiagonal = \
                self.vasicek_diagonals(sigma, kappa, theta,
                                       r_min, dr, N, dtau)
            v_mplus1, iterations = \
                self.iterate(subdiagonal, diagonal, superdiagonal,
                             v_mplus1, eex, max_policy_iter)
        return r, v_mplus1

    def vasicek_params(self, r0, M, sigma, kappa, theta, T,
```

```
                    prob, grid_struct_const=0.25, rs=None):
        (r_min, r_max) = (rs[0], rs[-1]) if not rs is None \
            else self.vasicek_limits(r0, sigma, kappa,
                                     theta, T, prob)
        dt = T/float(M)
        N = self.calculate_N(grid_struct_const, dt,
                             sigma, r_max, r_min)
        dr = (r_max-r_min)/(N-1)
        return r_min, dr, N, dt

    def calculate_N(self, max_structure_const, dt,
                    sigma, r_max, r_min):
        N = 0
        while True:
            N += 1
            grid_structure_interval = dt*(sigma**2)/(
                ((r_max-r_min)/float(N))**2)
            if grid_structure_interval > max_structure_const:
                break

        return N

    def vasicek_limits(self, r0, sigma, kappa,
                       theta, T, prob=1e-6):
        er = theta+(r0-theta)*math.exp(-kappa*T)
        variance = (sigma**2)*T if kappa==0 else \
                    (sigma**2)/(2*kappa)*(1-math.exp(-2*kappa*T))
        stdev = math.sqrt(variance)
        r_min = self.norminv(prob, er, stdev)
        r_max = self.norminv(1-prob, er, stdev)
        return r_min, r_max

    def vasicek_diagonals(self, sigma, kappa, theta,
                          r_min, dr, N, dtau):
        rn = np.r_[0:N]*dr + r_min
        subdiagonals = kappa*(theta-rn)*dtau/(2*dr) - \
                       0.5*(sigma**2)*dtau/(dr**2)
        diagonals = 1 + rn*dtau + sigma**2*dtau/(dr**2)
```

```
        superdiagonals = -kappa*(theta-rn)*dtau/(2*dr) - \
                         0.5*(sigma**2)*dtau/(dr**2)

        # Implement boundary conditions.
        if N > 0:
            v_subd0 = subdiagonals[0]
            superdiagonals[0] = superdiagonals[0] - \
                                subdiagonals[0]
            diagonals[0] += 2*v_subd0
            subdiagonals[0] = 0

        if N > 1:
            v_superd_last = superdiagonals[-1]
            superdiagonals[-1] = superdiagonals[-1] - \
                                 subdiagonals[-1]
            diagonals[-1] += 2*v_superd_last
            superdiagonals[-1] = 0

        return subdiagonals, diagonals, superdiagonals

    def check_exercise(self, V, eex):
        return V > eex

    def exercise_call_price(self, R, ratio, tau):
        K = ratio*np.exp(-R*tau)
        return K

    def vasicek_policy_diagonals(self, subdiagonal, diagonal,
                                 superdiagonal, v_old, v_new,
eex):
        has_early_exercise = self.check_exercise(v_new, eex)
        subdiagonal[has_early_exercise] = 0
        superdiagonal[has_early_exercise] = 0
        policy = v_old/eex
        policy_values = policy[has_early_exercise]
        diagonal[has_early_exercise] = policy_values
```

```
            return subdiagonal, diagonal, superdiagonal

        def iterate(self, subdiagonal, diagonal, superdiagonal,
                    v_old, eex, max_policy_iter=10):
            v_mplus1 = v_old
            v_m = v_old
            change = np.zeros(len(v_old))
            prev_changes = np.zeros(len(v_old))

            iterations = 0
            while iterations <= max_policy_iter:
                iterations += 1

                v_mplus1 = self.tridiagonal_solve(subdiagonal,
    diagonal,
                                                  superdiagonal,
    v_old)
                subdiagonal, diagonal, superdiagonal = \
                    self.vasicek_policy_diagonals(subdiagonal,
    diagonal,
                                                  superdiagonal,
    v_old,
                                                  v_mplus1, eex)

                is_eex = self.check_exercise(v_mplus1, eex)
                change[is_eex] = 1

                if iterations > 1:
                    change[v_mplus1 != v_m] = 1

                is_no_more_eex = False if True in is_eex else True
                if is_no_more_eex:
                    break

                v_mplus1[is_eex] = eex[is_eex]
                changes = (change == prev_changes)

                is_no_further_changes = all((x == 1) for x in changes)
                if is_no_further_changes:
```

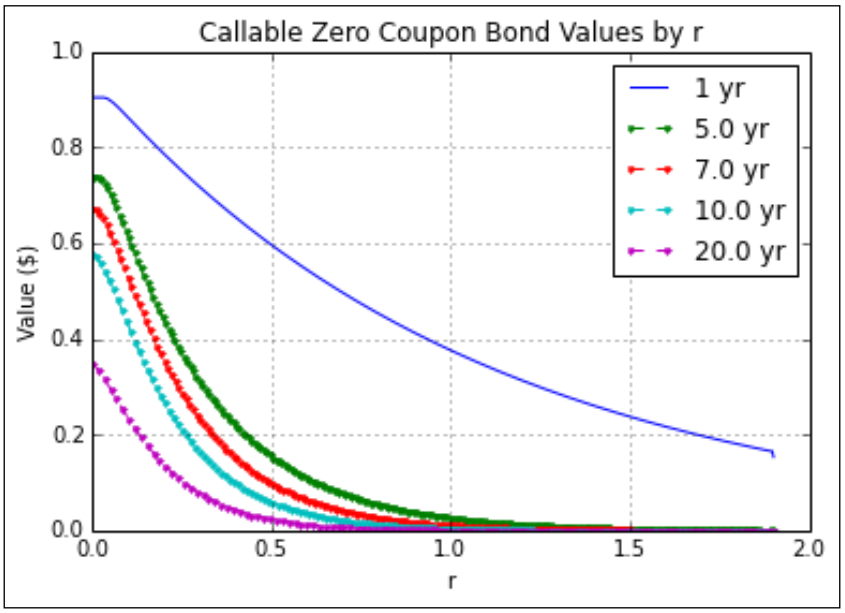
```
                break

            prev_changes = change
            v_m = v_mplus1

        return v_mplus1, (iterations-1)

    def tridiagonal_solve(self, a, b, c, d):
        nf = len(a)  # Number of equations
        ac, bc, cc, dc = map(np.array, (a, b, c, d))  # Copy the
array
        for it in xrange(1, nf):
            mc = ac[it]/bc[it-1]
            bc[it] = bc[it] - mc*cc[it-1]
            dc[it] = dc[it] - mc*dc[it-1]

        xc = ac
        xc[-1] = dc[-1]/bc[-1]

        for il in xrange(nf-2, -1, -1):
            xc[il] = (dc[il]-cc[il]*xc[il+1])/bc[il]

        del bc, cc, dc  # Delete variables from memory

        return xc
```

Assume that we run this model with the parameters: `r0` is `0.05`, `R` is `0.05`, `ratio` is `0.95`, `sigma` is `0.03`, `kappa` is `0.15`, `theta` is `0.05`, `prob` is `1e-6`, `M` is `250`, `max_policy_iter` is `10`, `grid_struc_interval` is `0.25`, and we are interested in the values of the interest rates between 0 percent and 2 percent. The following Python code demonstrates this model for maturities of 1 year, 5 years, 7 years, 10 years, and 20 years:

```
>>> r0 = 0.05
>>> R = 0.05
>>> ratio = 0.95
>>> sigma = 0.03
>>> kappa = 0.15
```

```
>>> theta = 0.05
>>> prob = 1e-6
>>> M = 250
>>> max_policy_iter=10
>>> grid_struct_interval = 0.25
>>> rs = np.r_[0.0:2.0:0.1]
>>>
>>> Vasicek = VasicekCZCB()
>>> r, vals = Vasicek.vasicek_czcb_values(r0, R, ratio, 1.,
...                                       sigma, kappa, theta,
...                                       M, prob,
...                                       max_policy_iter,
...                                       grid_struct_interval,
...                                       rs)
>>>
>>> import matplotlib.pyplot as plt
>>> plt.title("Callable Zero Coupon Bond Values by r")
>>> plt.plot(r, vals, label='1 yr')
>>>
>>> for T in [5., 7., 10., 20.]:
...    r, vals = \
...         Vasicek.vasicek_czcb_values(r0, R, ratio, T,
...                                     sigma, kappa,
                                        theta, M, prob,
...                                     max_policy_iter,
...                                     grid_struct_interval,
...                                     rs)
...     plt.plot(r, vals, label=str(T)+' yr',
...              linestyle="--", marker=".")
>>>
>>> plt.ylabel("Value ($)")
>>> plt.xlabel("r")
>>> plt.legend()
>>> plt.grid(True)
>>> plt.show()
```

After running the preceding commands, you get the following output:



We obtained the theoretical values of pricing callable zero-coupon bonds for various maturities for various interest rates.

Other considerations in callable bond pricing

In pricing callable zero-coupon bonds, we used the Vasicek interest rate process to model interest rate movement with the aid of a normal distribution process. We have earlier demonstrated that the Vasicek model can produce negative interest rates, which may not be practical for most economic cycles. Quantitative analysts often use more than one model in derivative pricing to obtain realistic results as much as possible. The CIR and Hull-White models are some of the commonly discussed models in financial studies. The limitation on these models is that they involve only one factor, or a single source of uncertainty.

We also looked at the implicit scheme of finite differences for policy iteration of the early exercise. Another method of consideration is the Crank-Nicolson method of finite differences. Other methods include the Monte Carlo simulation for calibration of this model.

Finally, we obtained a final list of short rates and callable bond prices. To infer a fair value of the callable bond for a particular short rate, interpolation of the list of bond prices is required. Often, the linear interpolation method is used. Other interpolation methods of consideration are the cubic and spline interpolation methods.

Summary

In this chapter, we focused on interest rate and related derivative pricing with Python. Most bonds, such as US Treasury bonds, pay a fixed amount of interest semi-annually, while other bonds may pay quarterly, or annually. It is a characteristic of bonds that their prices are closely related to current interest rate levels in an inversely related manner. The normal or positive yield curve, where long-term interest rates are higher than short-term interest rates, is said to be upward sloping. In certain economic conditions, the yield curve can be inverted and is said to be downward sloping.

A zero-coupon bond is a bond that pays no coupons during its lifetime, except on maturity when the principal or face value is repaid. We implemented a simple zero-coupon bond calculator in Python.

The yield curve can be derived from the short-term zero or spot rates of securities, such as zero-coupon bonds, T-bills, notes, and Eurodollar deposits using a bootstrapping process. Using Python, we used a lot of bond information to plot a yield curve, and derived forward rates, yield-to-maturity, and bond prices from the yield curve.

Two important metrics to bond traders are duration and convexity. Duration is a sensitivity measure of bond prices to yield changes. Convexity is the sensitivity measure of the duration of a bond to yield changes. We implemented calculations using the modified duration model and convexity calculator in Python.

Short rate models are frequently used in the evaluation of interest rate derivatives. Interest rate modeling is a fairly complex topic since they are affected by a multitude of factors, such as economic states, political decisions, government intervention, and the laws of supply and demand. A number of interest rate models have been proposed to account for various characteristics of interest rates. Some of the interest rate models we have discussed are the Vasicek model, CIR model, and Rendleman and Bartter model.

Bond issuers may embed options within a bond to allow them the right, but not the obligation, to buy or sell the issued bond at a predetermined price during a specified period of time. The price of a callable bond can be thought of as the price difference of a bond without an option and the price of the embedded call option. Using Python, we took a look at pricing a callable zero-coupon bond by applying the Vasicek model to the implicit method of finite differences. This method is, however, just one of the many methods that quantitative analysts use in bond options modeling.

In the next chapter, we will explore analytics with Python and VSTOXX.

6
Interactive Financial Analytics with Python and VSTOXX

Investors use volatility derivatives to diversify and hedge their risk in equity and credit portfolios. Since long-term investors in equity funds are exposed to downside risk, volatility can be used as a hedge for the tail risk and replacement for the put options. In the United States, the **Chicago Board Options Exchange** (**CBOE**) **Volatility Index** (**VIX**) measures the short-term volatility implied by S&P 500 stock index option prices. Many people around the world use the VIX to measure the stock market volatility over the next 30-day period. In Europe, the **EURO STOXX 50 Volatility** (**VSTOXX**) market index is based on the market prices of a basket of **Euro STOXX 50 Index Options** (**OESX**) and measures the implied market volatility over the next 30 days on the EURO STOXX 50 Index. For benchmark strategies utilizing the EURO STOXX 50 Index, the nature of its negative correlation with the VSTOXX presents a viable way of avoiding benchmark-rebalancing costs. The statistical nature of volatility allows traders to perform mean-reverting strategies, dispersion trading, and volatility spread trading, among others.

In this chapter, we will take a look at performing data analytics on the VSTOXX, EURO STOXX 50 Index, and OESX. The code presented here runs on the IPython Notebook, the interactive component of Python, to help us visualize data and study relationships between them.

In this chapter, we will discuss the following topics:

- Introduction to STOXX and the Eurex Exchange
- Introduction to the EURO STOXX 50 Index, VSTOXX, and VIX
- Gathering the EURO STOXX 50 Index and VSTOXX data with Python
- Using the `urllib` and `lxml` modules to read and traverse HTML data
- Understanding the file data format published by VSTOXX

- Performing financial analytics on the EURO STOXX 50 Index and VSTOXX
- Gathering the OESX data from the Eurex website
- Studying the formula used in calculating the VSTOXX sub-indexes
- Calculating the VSTOXX sub-indexes using OESX
- Studying the formula used in calculating the VSTOXX main index
- Calculating the VSTOXX main index from the VSTOXX sub-indexes
- Analyzing the results between the calculated values and actual values

Volatility derivatives

The two most popular volatility indexes worldwide are the VIX and VSTOXX, which are available in the United States and Europe respectively. The VSTOXX is based on OESX that trades on the Eurex Exchange. The Eurex Exchange website provides comprehensive information on the VSTOXX sub-indices and historical data, which we can analyze. We begin by understanding the background of these products before performing financial analytics on them in the later sections of this chapter.

STOXX and the Eurex

In the United States, the **Dow Jones Industrial Average** is one of the most widely watched stock market indexes, created of course by Dow Jones. In Europe, one such company is STOXX Limited.

Formed in 1997, STOXX Limited is headquartered in Zurich, Switzerland and calculates approximately 7,000 indices globally. As an index provider, it develops, maintains, distributes, and markets a comprehensive range of indices that are known to be strictly rule-based and transparent.

STOXX provides a number of equity indices in the categories: benchmark indices, blue-chip indices, dividend indices, size indices, sector indices, style indices, optimized indices, strategy indices, theme indices, sustainability indices, faith-based indices, smart beta indices, and calculation products.

The Eurex Exchange is a derivatives exchange in Frankfurt, Germany offering more than 1,900 products, including equity indices, futures, options, ETFs, dividends, bonds, and repos. Many of STOXX's products and derivatives trade on the Eurex.

The EURO STOXX 50 Index

Designed by STOXX Limited, the EURO STOXX 50 Index is one of the most liquid stock indexes worldwide, serving many indices products listed on the Eurex. It was introduced on February 26 1998 and is made up of 50 blue-chip stocks from the 12 Eurozone countries: Austria, Belgium, Finland, France, Germany, Greece, Ireland, Italy, Luxembourg, the Netherlands, Portugal, and Spain. The EURO STOXX 50 Index futures and options contracts are available and traded on the Eurex Exchange. Recalculation of the index takes place typically every 15 seconds based on real-time prices.

The ticker symbol for the EURO STOXX 50 Index is SX5E. EURO STOXX 50 Index Options take on the ticker symbol OESX. We may come across these symbols when working with EURO STOXX 50 Index data in the later sections of this chapter.

The VSTOXX

The VSTOXX or EURO STOXX 50 Volatility is a class of volatility derivatives serviced by the Eurex Exchange. The VSTOXX market index is based on the market prices of a basket of OESX quoted at-the-money or out-of-the-money. It measures the implied market volatility over the next 30 days on the EURO STOXX 50 Index.

Volatility derivatives are tradable products whose payoff depends on the volatility of the underlying assets. Examples of such products are volatility swaps and variance swaps.

Investors use volatility derivatives for benchmark strategies utilizing the EURO STOXX 50 Index, the nature of its negative correlation with the VSTOXX presents a viable way of avoiding benchmark-rebalancing costs. The statistical nature of volatility allows traders to perform mean-reverting strategies, dispersion trading, and volatility spread trading, among others. Recalculation of the index takes place every 5 seconds.

The ticker symbol for the VSTOXX is V2TX. VSTOXX Options and VSTOXX Mini Futures based on the VSTOXX Index trades on the Eurex Exchange.

The VIX

Like the STOXX, the CBOE Volatility Index (VIX) measures the short-term volatility implied by S&P 500 stock index option prices. Many people around the world think of the VIX to be a popular measurement tool for the stock market volatility over the next 30-day period. The VIX recalculates every 15 seconds.

VIX Options and VIX Futures are based on the VIX and trades on the CBOE.

Gathering the EUROX STOXX 50 Index and VSTOXX data

STOXX Limited publishes the historical daily end-of-day index prices on their website at `http://www.stoxx.com/data/historical/historical_benchmark.html`.

The "STOXX Europe 600 (all regions)" historical daily data can be obtained at `http://www.stoxx.com/download/historical_values/hbrbcpe.txt` under **Benchmark Indices** of the **Historical Data** category of the website. The EURO STOXX 50 Index can be found within this data file.

The VSTOXX historical daily data can be obtained at `http://www.stoxx.com/download/historical_values/h_vstoxx.txt`. The EURO STOXX 50 Volatility link is found under **Strategy Indices** of the **Historical Data** category of the website.

The Python module `urllib` can be used to interact with the web resources through the `urlretrieve` function to download data from an external source onto our local disk.

The following Python code lets us download the required data text files onto our destination directory defined by the `data_folder` variable. In this example, the folder named `data` is used. If this folder does not exist in your working directory, create one now before running the codes:

```
from urllib import urlretrieve

url_path = 'http://www.stoxx.com/download/historical_values/'
stoxxeu600_url = url_path + 'hbrbcpe.txt'
vstoxx_url = url_path + 'h_vstoxx.txt'

data_folder = 'data/'  # Save file to local target destination.

stoxxeu600_filepath = data_folder + "stoxxeu600.txt"
vstoxx_filepath = data_folder + "vstoxx.txt"
```

Now, we will run the following command to download the STOXX Europe 600 Index data file:

```
>>> urlretrieve(stoxxeu600_url, stoxxeu600_filepath)
('data/stoxxeu600.txt', <httplib.HTTPMessage instance at 0x105b47290>)
```

We can do the same to download the VSTOXX data file:

```
>>> urlretrieve(vstoxx_url, vstoxx_filepath)
('data/vstoxx.txt', <httplib.HTTPMessage instance at 0x105c764d0>)
```

To check whether our data has been downloaded successfully to our hard disk, run the following command:

```
>>> import os.path
>>> os.path.isfile(stoxxeu600_filepath)
True
>>> os.path.isfile(vstoxx_filepath)
True
```

That's right, the file now exists in our directory. Otherwise, the output will be `False`.

With the STOXX Europe 600 data file in hand, let's see what the first five lines of the text file might look like:

```
>>> with open(stoxxeu600_filepath, 'r') as opened_file:
...     for i in range(5):
...         print opened_file.readline(),
Price Indices - EURO Currency
Date    ;Blue-Chip;Blue-Chip;Broad    ; Broad   ;Ex UK    ;Ex Euro
Zone;Blue-Chip; Broad
        ;  Europe ;Euro-Zone;Europe   ;Euro-Zone;         ;            ;
Nordic  ; Nordic
        ;  SX5P   ;  SX5E   ;SXXP     ;SXXE     ; SXXF    ;    SXXA    ;
DK5F ; DKXF
31.12.1986;775.00 ;  900.82 ;   82.76 ;   98.58 ;   98.06 ;   69.06 ;
645.26  ;  65.56
```

From the previous output, we can see that semicolons separate the data in the STOXX Europe 600 text file. In the last of the top four rows of information lie our headers of interest, being `Date`, `SX5P`, `SX5E`, `SXXP`, `SXXE`, `SXXF`, `SXXA`, `DK5F`, and `DKXF`. With this information, we can begin to parse the data with pandas into a `DataFrame` object, as explained in the following code:

```
import pandas as pd

columns = ['Date', 'SX5P', 'SX5E', 'SXXP', 'SXXE',
           'SXXF', 'SXXA', 'DK5F', 'DKXF', 'EMPTY']
stoxxeu600 = pd.read_csv(stoxxeu600_filepath,
                 index_col=0,
                 parse_dates=True,
                 dayfirst=True,
                 header=None,
                 skiprows=4,
                 names=columns,
```

```
                    sep=';'
                    )
    del stoxxeu600['EMPTY']
```

Here, we added an extra `EMPTY` column to the account for the trailing semicolons found at certain rows in the data. The extra column information is deleted after the parsing is done.

The `read_csv` function is a nifty pandas function that parses and converts a file into a pandas `DataFrame` object. A `DataFrame` object is a two-dimensional data structure very much like a table. The extra arguments tells us to treat the first column values as date objects, ignore the top four rows, parse the data with semicolon separators, and introduce the column names as defined in our `columns` variable. The `stoxx50` variable now takes on the pandas `DataFrame` data type. To view more details about our new `DataFrame` object, we can use the `info` function of Pandas as follows:

```
>>> stoxxeu600.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 7189 entries, 1986-12-31 00:00:00 to 2014-11-17 00:00:00
Data columns (total 8 columns):
SX5P    7189 non-null float64
SX5E    7189 non-null float64
SXXP    7189 non-null float64
SXXE    7189 non-null float64
SXXF    7189 non-null float64
SXXA    7189 non-null float64
DK5F    7189 non-null float64
DKXF    7189 non-null float64
dtypes: float64(8)
```

The column definitions of the STOXX Europe 600 data file are given in the following table:

Abbreviation	Benchmark index
SX5P	STOXX Europe 50
SX5E	Euro STOXX 50 Index
SXXP	STOXX Europe 600
SXXE	EURO STOXX
SXXF	STOXX Europe 600 ex UK
SXXA	STOXX Europe 600 ex Eurozone

Abbreviation	Benchmark index
DK5F	STOXX Nordic 30
DKXF	STOXX Nordic

Now, let's do the same for the VSTOXX data file:

```
>>> with open(vstoxx_filepath, 'r') as opened_file:
...     for i in range(5):
...         print opened_file.readline(),
EURO STOXX 50 Volatility Indices,,,,,,,,,
,VSTOXX,Sub-Index 1M,Sub-Index 2M,Sub-Index 3M,Sub-Index 6M,Sub-Index
9M,Sub-Index 12M,Sub-Index 18M,Sub-Index 24M
Date,V2TX,V6I1,V6I2,V6I3,V6I4,V6I5,V6I6,V6I7,V6I8
04.01.1999,18.2033,21.2458,17.5555,31.2179,33.3124,33.7327,33.2232,31.853
5,23.8209
05.01.1999,29.6912,36.6400,28.4274,32.6922,33.7326,33.1724,32.8457,32.290
4,25.0532
```

From the preceding output information, we can see that the VSTOXX data file is slightly different from the STOXX Europe 600 data file. Data in the VSTOXX text file is separated by commas with the first two rows carrying the additional information, which we will discard.

In the same fashion, we will do the same for the STOXX Europe 600 data file. We will parse the VSTOXX data text file to a pandas `DataFrame` object:

```
vstoxx = pd.read_csv(vstoxx_filepath,
                     index_col=0,
                     parse_dates=True,
                     dayfirst=True,
                     header=2)
```

In the later sections of this chapter, we will read the VSTOXX data again. For easy access to the VSTOXX data, let's save the data file as a CSV file in the `data` folder of our working directory with the name `vstoxx.csv`:

```
>>> vstoxx.to_csv('data/vstoxx.csv')
```

The `vstoxx` variable is now a pandas `DataFrame` object type, and we can use the `info` function of pandas to peek at its properties:

```
>>> print vstoxx.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 4046 entries, 1999-01-04 00:00:00 to 2014-11-17 00:00:00
```

```
Data columns (total 9 columns):
V2TX    4046 non-null float64
V6I1    3625 non-null float64
V6I2    4046 non-null float64
V6I3    3995 non-null float64
V6I4    4046 non-null float64
V6I5    4046 non-null float64
V6I6    4031 non-null float64
V6I7    4046 non-null float64
V6I8    4035 non-null float64
dtypes: float64(9)
```

The column definitions of the VSTOXX data file are given in the following table:

Abbreviation	Index
V2TX	The actual EURO STOXX 50 Volatility values
V6I1	VSTOXX 1 month
V6I2	VSTOXX 2 months
V6I3	VSTOXX 3 months
V6I4	VSTOXX 6 months
V6I5	VSTOXX 9 months
V6I6	VSTOXX 12 months
V6I7	VSTOXX 18 months
V6I8	VSTOXX 24 months

Merging the data

Since the earliest dates in the text files are `31.12.1986` and `04.01.1999` for the STOXX Europe 600 and VSTOXX data file respectively, we will require both the datasets to begin from a common date at `04.01.1999`. We will also use values from the `SX5E` and `V2TX` columns to retrieve our EURO STOXX 50 Index and VSTOXX historical data values. The following Python code extracts these values into a new Pandas `DataFrame` object:

```
import datetime as dt

cutoff_date = dt.datetime(1999, 1, 4)
data = pd.DataFrame(
{'EUROSTOXX' :stoxxeu600['SX5E'][stoxxeu600.index >= cutoff_date],
 'VSTOXX':vstoxx['V2TX'][vstoxx.index >= cutoff_date]})
```

Now, let's take a look at our `DataFrame` information:

```
>>> print data.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 4072 entries, 1999-01-04 00:00:00 to 2014-11-18 00:00:00
Data columns (total 2 columns):
EUROSTOXX    4071 non-null float64
VSTOXX       4046 non-null float64
dtypes: float64(2)
```

Also, let's take a look at the top five lines of our new `DataFrame` object:

```
>>> print data.head(5)
            EUROSTOXX   VSTOXX
Date
1999-01-04    3543.10  18.2033
1999-01-05    3604.67  29.6912
1999-01-06    3685.36  25.1670
1999-01-07    3627.87  32.5205
1999-01-08    3616.57  33.2296
```

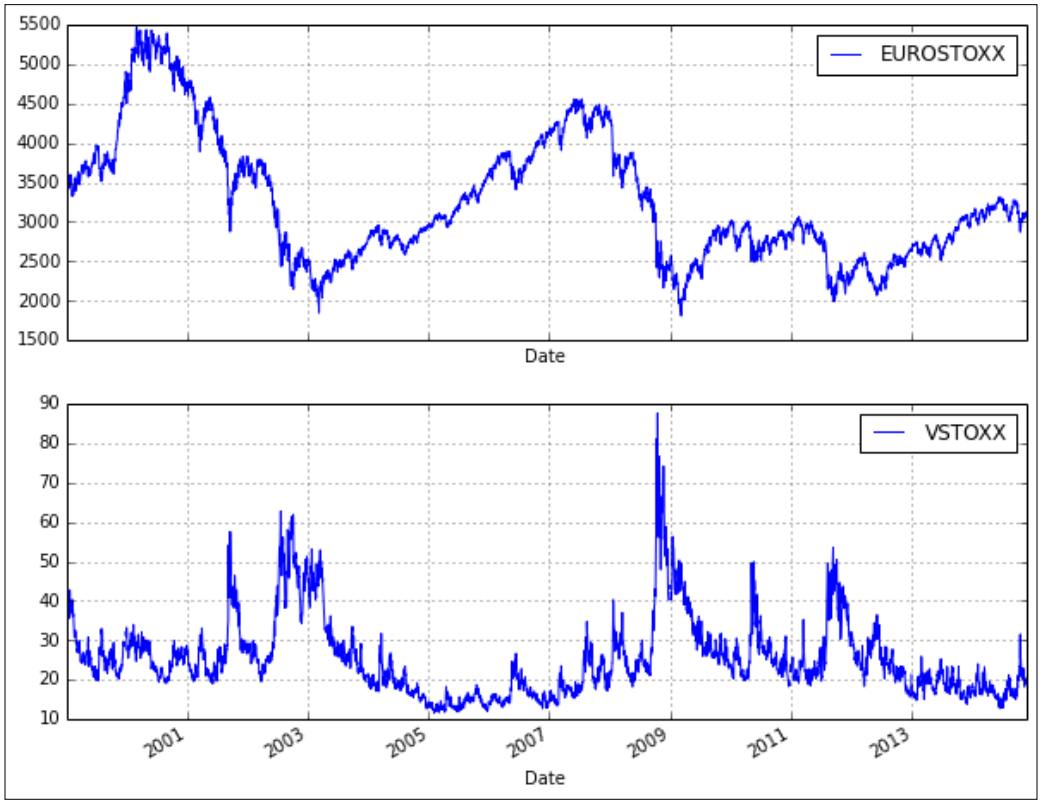
Financial analytics of SX5E and V2TX

Another nifty function of pandas is the `describe` function that gives us a summary statistics of every value inside each column of the Pandas `DataFrame` object:

```
>>>  print data.describe()
         EUROSTOXX       VSTOXX
count  4072.000000  4048.000000
mean   3254.538183    25.305428
std     793.191950     9.924404
min    1809.980000    11.596600
25%    2662.460000    18.429500
50%    3033.880000    23.168600
75%    3753.542500    28.409550
max    5464.430000    87.512700
[8 rows x 2 columns]
```

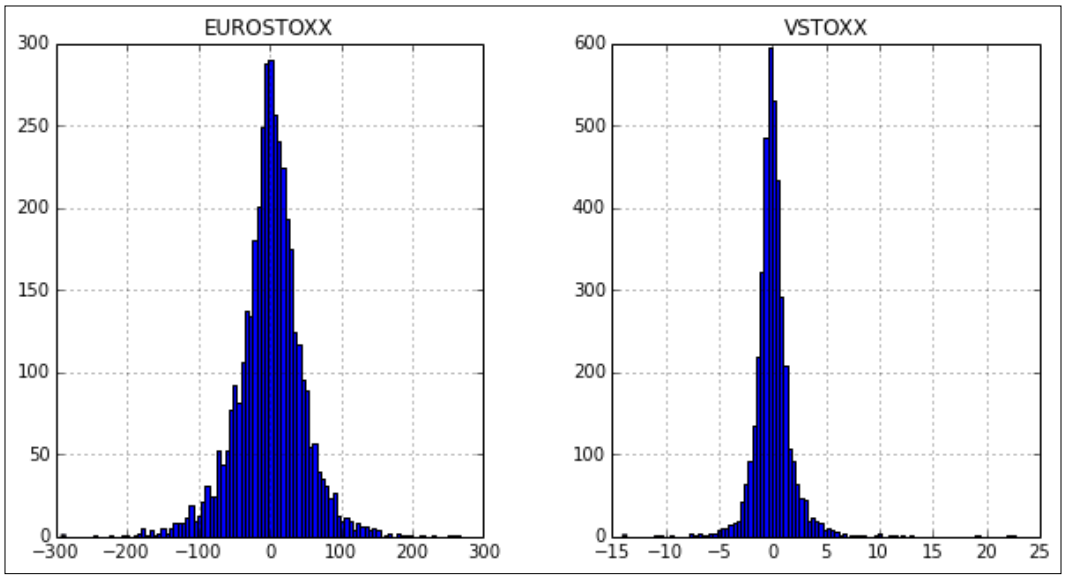
Pandas allows the values in the `DataFrame` object to be visualized as a graph using the `plot` function. Let's plot the EURO STOXX 50 and VSTOX to see how they look like over the years:

```
>>> from pylab import *
>>> data.plot(subplots=True,
...           figsize=(10, 8),
...           color="blue",
...           grid=True)
>>> show()
Populating the interactive namespace from numpy and matplotlib
array([<matplotlib.axes.AxesSubplot object at 0x10f4464d0>,
<matplotlib.axes.AxesSubplot object at 0x10f4feed0>], dtype=object)
```



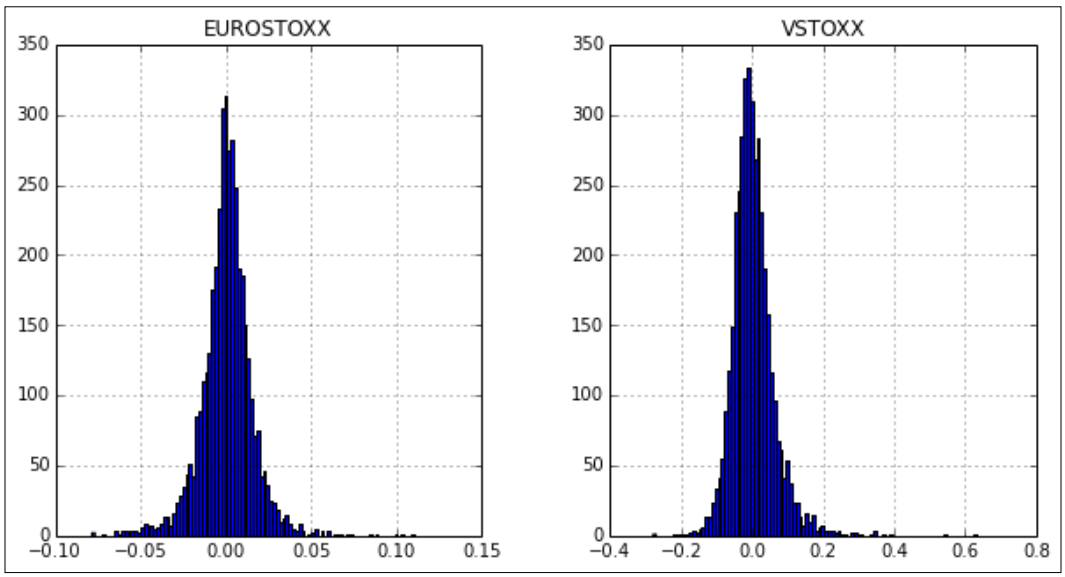
Perhaps we might be interested in the daily returns of both the indexes. The `diff` method returns the set of differences between the prior period values. A histogram can be used to give us a rough sense of the data density estimation over a bin interval of 100:

```
>>> data.diff().hist(figsize=(10, 5),
...                  color='blue',
...                  bins=100)
array([[<matplotlib.axes.AxesSubplot object at 0x11083d910>,
<matplotlib.axes.AxesSubplot object at 0x110f3f7d0>]], dtype=object)
```



The same effect can also be achieved with the `pct_change` function that gives us the percentage change over the prior period values:

```
>>> data.pct_change().hist(figsize=(10, 5),
...                        color='blue',
...                        bins=100)
array([[<matplotlib.axes.AxesSubplot object at 0x11132b810>,
<matplotlib.axes.AxesSubplot object at 0x111ef1f90>]], dtype=object)
```

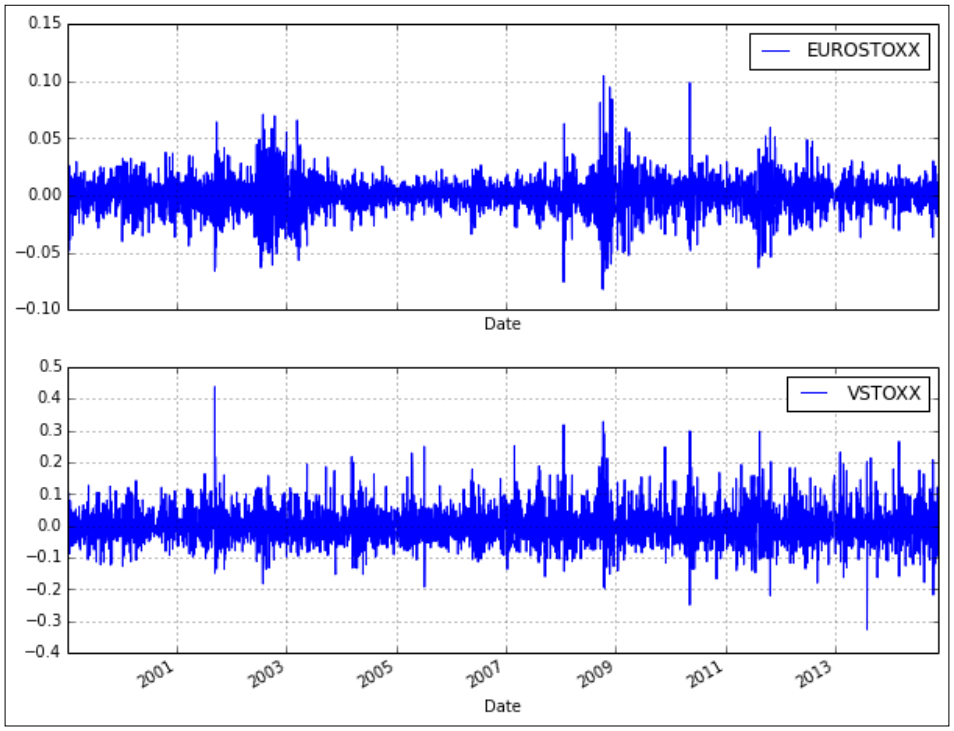


For quantitative analysis of returns, we are interested in the logarithm of daily returns. Why use log returns over simple returns? There are several reasons, but the most important of them is normalization, and this avoids the problem of negative prices.

We can use the `shift` function of pandas to shift the values by a certain number of periods. The `dropna` function removes the unused values at the end of the logarithm calculation transformation. The `log` function of NumPy helps you calculate the logarithm of all values in the `DataFrame` object as a vector and will be stored in the `log_returns` variable as a `DataFrame` object. The logarithm values can then be plotted in the same way as we did earlier, to give us a graph of daily log returns. Here is the code to plot the logarithm values:

```
>>> from pylab import *
>>> import numpy as np
>>>
```

```
>>> log_returns = np.log(data / data.shift(1)).dropna()
>>> log_returns.plot(subplots=True,
...                  figsize=(10, 8),
...                  color='blue',
...                  grid=True)
>>> show()
Populating the interactive namespace from numpy and matplotlib
array([<matplotlib.axes.AxesSubplot object at 0x11553f1d0>,
<matplotlib.axes.AxesSubplot object at 0x117c15990>], dtype=object)
```



Correlation between SX5E and V2TX

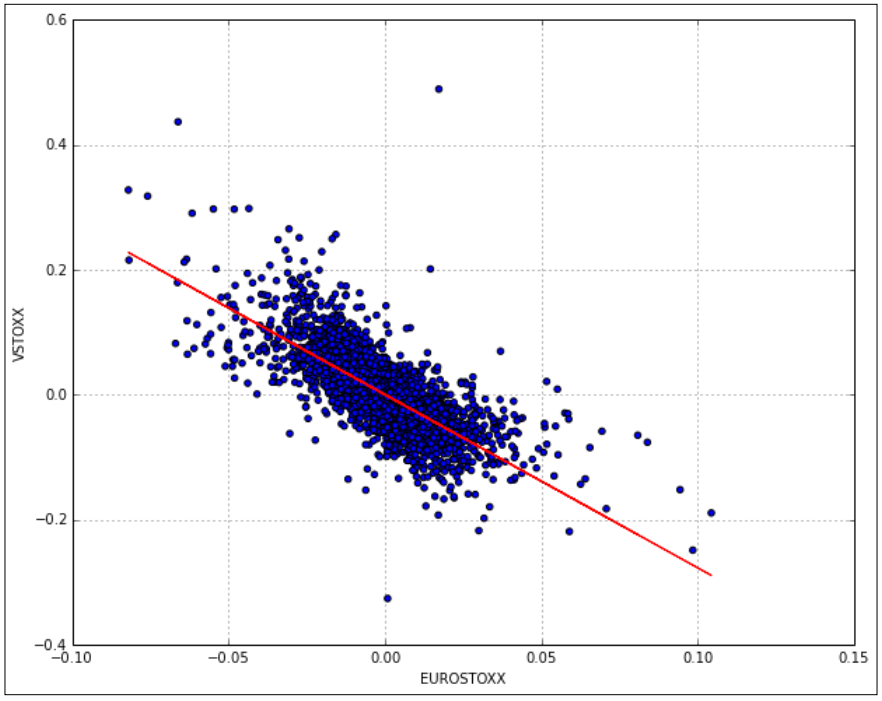
We can use the `corr` function to derive the correlation values between each column of values in the pandas `DataFrame` object, as in the following Python example:

```
>>> print log_returns.corr()
            EUROSTOXX    VSTOXX
EUROSTOXX    1.000000 -0.732545
```

```
VSTOXX     -0.732545  1.000000
[2 rows x 2 columns]
```

At `-0.7325`, the EURO STOXX 50 Index is negatively correlated with the STOXX. To help us better visualize this relationship, we can plot both the sets of the daily log return values as a scatter plot. The `statsmodels.api` module is used to obtain the ordinary least squares regression line between the scattered data:

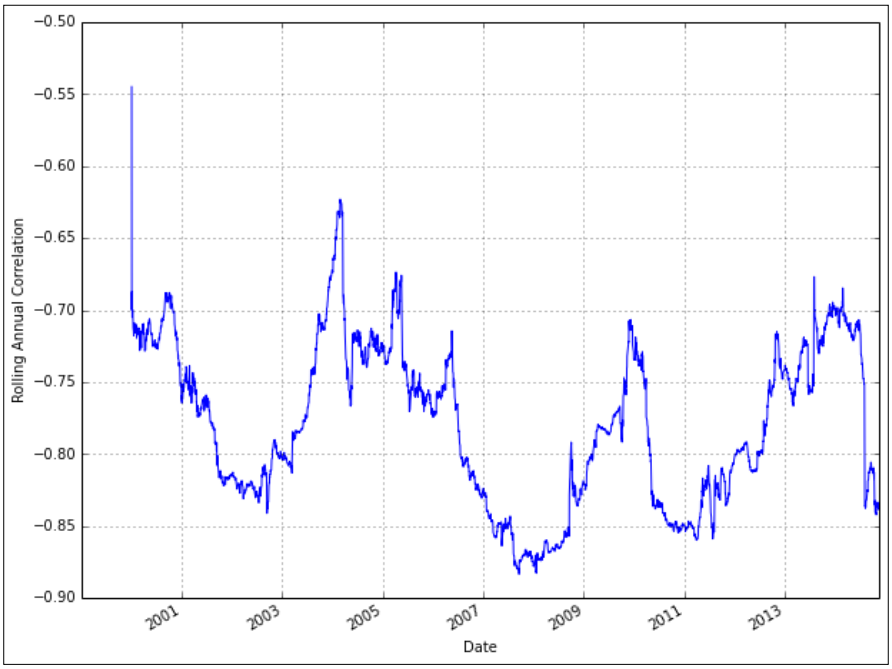
```
>>> import statsmodels.api as sm
>>>
>>> log_returns.plot(figsize=(10,8),
...                  x="EUROSTOXX",
...                  y="VSTOXX",
...                  kind='scatter')
>>>
>>> ols_fit = sm.OLS(log_returns['VSTOXX'].values,
...              log_returns['EUROSTOXX'].values).fit()
>>>
>>> plot(log_returns['EUROSTOXX'], ols_fit.fittedvalues, 'r')
[<matplotlib.lines.Line2D at 0x117704550>]
```



The downward-sloping regression line, as shown in the preceding graph, confirms the negative correlation relationship between the EURO STOXX 50 and the VSTOXX indices.

The `rolling_corr` function of pandas computes the moving-window correlation between two time series over time. We will use a value of 252 to represent the number of trading days in the moving window to compute the annual rolling correlation, using the following commands:

```
>>> pd.rolling_corr(log_returns['EUROSTOXX'],
...                 log_returns['VSTOXX'],
...                 window=252).plot(figsize=(10,8))
>>> plt.ylabel('Rolling Annual Correlation')
<matplotlib.text.Text at 0x118feb810>
```



It can be seen from the preceding graph that the EURO STOXX 50 Index and VSTOXX are negatively correlated during the entire lifetime of the indices using 252 trading days per year.

Calculating the VSTOXX sub-indices

The VSTOXX data file `vstoxx.txt`, which we downloaded earlier, consists of eight sub-indexes that represent the calculated volatility index from the EURO STOXX 50 options that expire in 1, 2, 3, 6, 9, 12, 18, and 24 months. The VSTOXX Index represents the volatility outlook for the next 30 days, and it is the option series with the nearest expiry date over the next 30 days that is taken into account for the calculation of the VSTOXX Index.

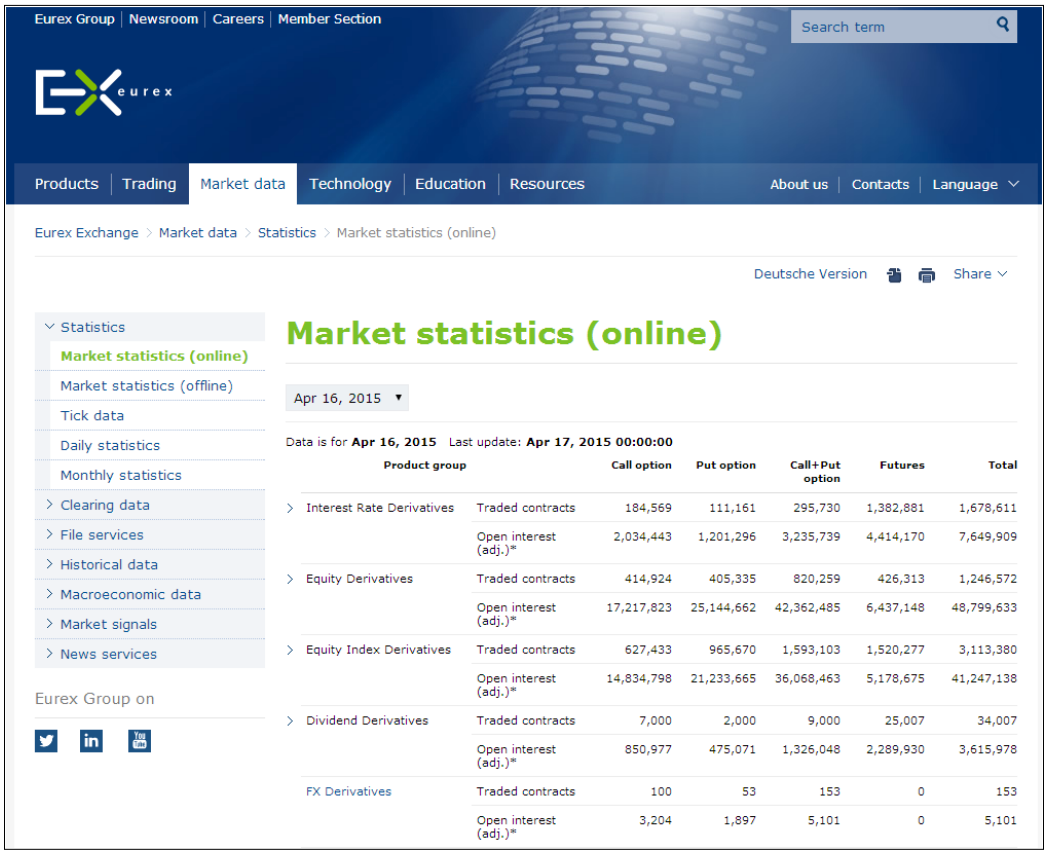
To help us examine the movement of the VSTOXX Index, we would need to study the movement of its component sub-index. To do so, we would need to refer to the OESX calls and put prices listed in the Eurex Exchange.

Getting the OESX data

The Eurex Exchange website contains the daily historical call and put options prices for the past 30 days. Unfortunately, there is no direct method of downloading and obtaining the data directly. A Python utility function is needed to scrape data off the web page and store it in a pandas `DataFrame` object for our analysis. The data can be obtained from

`http://www.eurexchange.com/exchange-en/market-data/statistics/market-statistics-online`.

A screenshot of the option prices web page is given as follows:



In the **Market statistics (online)** page, navigate to **Equity Index Derivatives**, then **Blue Chip**. Click on **OESX**. Some key information is presented on this page. Firstly, the drop-down box contains a list of available dates to choose from. Secondly, it contains the last updated date and time of the prices on the page. Thirdly, it contains a table that shows the option of the expiry month for the selected date. Selecting an option type and expiry month brings us to a page containing a table of the daily option prices.

The following screenshot shows the call prices for the selected date. The put prices are contained in a separate link:

Product ISIN

Call | Put

Strike price	Version number	Opening price	Daily High	Daily Low	Underlying closing price	Daily settlem. price	Traded contracts	Open Interest (adj.)*
500.00	0.00	0.00	0.00	0.00	0.00	2,690.00	0	4,481
550.00	0.00	0.00	0.00	0.00	0.00	2,640.00	0	0
600.00	0.00	0.00	0.00	0.00	0.00	2,590.00	200	8,064
650.00	0.00	0.00	0.00	0.00	0.00	2,540.00	0	0
700.00	0.00	0.00	0.00	0.00	0.00	2,490.00	0	890

The table contains comprehensive information about the option prices. The two columns that we are interested in are **Strike price** and **Daily settlem. price**. From the values of these two columns, we can then derive the hypothetical sub-index value for the chosen expiry month.

Formulas to calculate the VSTOXX sub-index

The *STOXX Strategy Index Guide* is available on the STOXX website at `http://www.stoxx.com/download/indices/rulebooks/stoxx_strategy_guide.pdf`. This document contains details on the formulas used to calculate its indexes in the Eurex system.

The value of the VSTOXX sub-index is given as:

$$V = 100\sqrt{\sigma^2}$$

Here:

$$\sigma^2 = \frac{2}{T_i / T_{365}} \sum_j \frac{\Delta K_{i,j}}{K_{i,j}^{\ 2}} R_i M\left(K_{i,j}\right) - \frac{2}{T_i / T_{365}} \left[\frac{F_i}{K_{i,0}} - 1\right]^2$$

T_i is the time to maturity of the ith OESX instrument in terms of seconds, and T_{365} is the number of seconds in a 365-day year.

$K_{i,0}$ is the highest strike price that does not exceed the forward price F_i.

F_i is the forward at-the-money price calculated from the price of the *ith* OESX expiry date, where the absolute difference between the call prices (*C*) and put prices (*P*) is the smallest and can be written as:

$$F_i = K_{\min|C-P|} + R_i\left(C - P\right)$$

Should multiple identical price differences exist, $K_{i,0}$ will be the closest strike price below the average of these forward prices.

$\Delta K_{i,j}$ is the mean distance between the lower and upper strike prices of $K_{i,j}$. At the maximum and minimum strike price boundaries, $\Delta K_{i,j}$ is taken to be the difference of the highest and second highest strike price. This value can also be written as:

$$\Delta K_i = \begin{cases} K_1 - K_0, & for\ i = 0 \\ \dfrac{K_{i+1} - K_{i-1}}{2}, & for\ i = 1\ to\ n-1 \\ K_n - K_{n-1}, & for\ i = n \end{cases}$$

R_i is the continuously compounded interest rate of the time remaining to maturity and can be written as:

$$R_i = e^{r_i \frac{T_i}{T_{365}}}$$

Here r_i is the interpolated interest rate available for the *ith* OESX expiry date.

$M\left(K_{i,j}\right)$ is the price of the out-of-the-money option. This value takes on the put prices for the strike prices below $K_{i,0}$ and call prices for the strike prices above $K_{i,0}$. At $K_{i,0}$, this value is the average of the sum of the call and put prices. This can also be written as:

$$M\left(K_{i,j}\right) = \begin{cases} P_1, & for\ K_{i,j} < K_{i,0} \\ \dfrac{P_i - C_i}{2}, & for\ K_{i,j} = K_{i,0} \\ C_v & for\ K_{i,j} > K_{i,0} \end{cases}$$

Implementation of the VSTOXX sub-index value

Let's create the classes that will help us read data from the Eurex web page, parse the data, calculate the sub-index values, and save them in a CSV file.

The `OptionUtility` class contains a number of utility methods that will help us perform the date and time conversion functions between strings and Python date objects. The various `OptionUtility` classes are explained as follows:

- The `VSTOXXCalculator` class contains the `calculate_sub_index` method that implements the formulas for calculating the sub-index for a particular option series.
- The `EurexWebPage` class contains methods for interacting with the data on the Eurex web page. The `lxml` Python module is required. It can be obtained at `http://lxml.de`.
- The `VSTOXXSubIndex` class contains the following methods for fetching the data and calculating the sub-indexes of an external file. The source code for all the classes are given at the end of this section:
    - `__init__(self, path_to_subindexes)`: This method contains the initialization of object instances used throughout this class. The final output values are stored in a CSV file, which is indicated by the `path_to_subindexes` variable.
    - `start(self, months=2, r=0.015)`: This is the main method to begin the process of downloading and calculating the data. By default, we are interested in calculating the sub-index, where the options expire in 2 months. An interest rate of 1.5 percent is assumed. A `for` loop is used to process each particular historical data iteratively. Using the `print` function within the `for` loop, helps us track our progress since it could take a while to read and calculate all the required sub-indexes.
    - `calculate_and_save_sub_indexes(self, selected_date, months_fwd, r)`: This method takes in a date selected from the drop-down list and fetches the option series data for the next month onwards till the expiry month, which is given by the `months_fwd` variable. For every expiry month information fetched, the sub-index is calculated and saved in a CSV file.

- `save_vstoxx_sub_index_to_csv(self, current_dt, sub_index, month)`: This method saves a single sub-index value for a single expiry month for a single trading day in a CSV file in the form of a pandas `DataFrame` object. If the `DataFrame` object does not exist, one is created. Otherwise, the existing data is appended to `DataFrame` and saved.
- `get_data(self, current_dt, expiry_dt)`: This method fetches the call and put option series data separately, which is then combined into a single pandas `DataFrame` object. The dataset and the time of data, as shown in the web page, is returned.

To run the program, we simply call the `start` method to collect the historical option series data that expires in 2 months:

```
>>> vstoxx_subindex = VSTOXXSubIndex(
...     "data/vstoxx_sub_indexes.csv")
>>> vstoxx_subindex.start(2)
Collecting historical data for 20141030 ...
Collecting historical data for 20141031 ...
Collecting historical data for 20141103 ...
…
Collecting historical data for 20141126 ...
Completed.
```

The data will be saved in `data/vstoxx_sub_indexes.csv` in our working directory folder.

The following Python code is the full implementation of all the classes:

```
import calendar as cal
import datetime as dt


class OptionUtility(object):

    def get_settlement_date(self, date):
        """ Get third friday of the month """
        day = 21 - (cal.weekday(date.year, date.month, 1) + 2) % 7
        return dt.datetime(date.year, date.month, day, 12, 0, 0)

    def get_date(self, web_date_string, date_format):
        """  Parse a date from the web to a date object """
```

```
            return dt.datetime.strptime(web_date_string, date_format)

        def fwd_expiry_date(self, current_dt, months_fws):
            return self.get_settlement_date(
                current_dt + relativedelta(months=+months_fws))



    import math


    class VSTOXXCalculator(object):

        def __init__(self):
            self.secs_per_day = float(60*60*24)
            self.secs_per_year = float(365*self.secs_per_day)

        def calculate_sub_index(self, df, t_calc, t_settle, r):
            T = (t_settle-t_calc).total_seconds()/self.secs_per_year
            R = math.exp(r*T)

            # Calculate dK
            df["dK"] = 0
            df["dK"][df.index[0]] = df.index[1]-df.index[0]
            df["dK"][df.index[-1]] = df.index[-1]-df.index[-2]
            df["dK"][df.index[1:-1]] = (df.index.values[2:]-
                                        df.index.values[:-2])/2
            # Calculate the forward price
            df["AbsDiffCP"] = abs(df["Call"]-df["Put"])
            min_val = min(df["AbsDiffCP"])
            f_df = df[df["AbsDiffCP"]==min_val]
            fwd_prices = f_df.index+R*(f_df["Call"]-f_df["Put"])
            F = np.mean(fwd_prices)

            # Get the strike not exceeding forward price
            K_i0 = df.index[df.index <= F][-1]

            # Calculate M(K(i,j))
            df["MK"] = 0
            df["MK"][df.index < K_i0] = df["Put"]
            df["MK"][K_i0] = (df["Call"][K_i0]+df["Put"][K_i0])/2.
            df["MK"][df.index > K_i0] = df["Call"]
```

```
        # Apply the variance formula to get the sub-index
        summation = sum(df["dK"]/(df.index.values**2)*R*df["MK"])
        variance = 2/T*summation-1/T*(F/float(K_i0)-1)**2
        subindex = 100*math.sqrt(variance)
        return subindex



import urllib
from lxml import html


class EurexWebPage(object):

    def __init__(self):
        self.url = "%s%s%s%s%s" % (
            "http://www.eurexchange.com/",
            "exchange-en/market-data/statistics/",
            "market-statistics-online/180102!",
            "onlineStats?productGroupId=846&productId=19068",
            "&viewType=3")
        self.param_url = "&cp=%s&month=%s&year=%s&busDate=%s"
        self.lastupdated_dateformat = "%b %d, %Y %H:%M:%S"
        self.web_date_format = "%Y%m%d"
        self.__strike_price_header__ = "Strike price"
        self.__prices_header__ = "Daily settlem. price"
        self.utility = OptionUtility()

    def get_available_dates(self):
        html_data = urllib.urlopen(self.url).read()
        webpage = html.fromstring(html_data)

        # Find the dates available on the website
        dates_listed = webpage.xpath(
            "//select[@name='busDate']" +
            "/option")

        return [date_element.get("value")
                for date_element in reversed(dates_listed)]

    def get_date_from_web_date(self, web_date):
        return self.utility.get_date(web_date,
                                     self.web_date_format)
```

```
    def get_option_series_data(self, is_call,
                               current_dt, option_dt):
        selected_date = current_dt.strftime(self.web_date_format)
        option_type = "Call" if is_call else "Put"
        target_url = (self.url +
                      self.param_url) % (option_type,
                                         option_dt.month,
                                         option_dt.year,
                                         selected_date)
        html_data = urllib.urlopen(target_url).read()
        webpage = html.fromstring(html_data)
        update_date = self.get_last_update_date(webpage)
        indexes = self.get_data_headers_indexes(webpage)
        data = self.__get_data_rows__(webpage,
                                      indexes,
                                      option_type)
        return data, update_date

    def __get_data_rows__(self, webpage, indexes, header):
        data = pd.DataFrame()
        for row in webpage.xpath("//table[@class='dataTable']/" +
                                 "tbody/tr"):
            columns = row.xpath("./td")
            if len(columns) > max(indexes):
                try:
                    [K, price] = \
                        [float(columns[i].text.replace(",",""))
                         for i in indexes]
                    data.set_value(K, header, price)
                except:
                    continue
        return data

    def get_data_headers_indexes(self, webpage):
        table_headers = webpage.xpath(
            "//table[@class='dataTable']" + \
            "/thead/th/text()")
        indexes_of_interest = [
            table_headers.index(
                self.__strike_price_header__),
            table_headers.index(
                self.__prices_header__)]
        return indexes_of_interest
```

```
    def get_last_update_date(self, webpage):
        return dt.datetime.strptime(webpage.
                                    xpath("//p[@class='date']/b")
                                    [-1].text,
                                    self.lastupdated_dateformat)


import pandas as pd

from dateutil.relativedelta import relativedelta
import numpy as np
import thread

class VSTOXXSubIndex:

    def __init__(self, path_to_subindexes):
        self.sub_index_store_path = path_to_subindexes
        self.utility = OptionUtility()
        self.webpage = EurexWebPage()
        self.calculator = VSTOXXCalculator()
        self.csv_date_format = "%m/%d/%Y"

    def start(self, months=2, r=0.015):
        # For each date available, fetch the data
        for selected_date in self.webpage.get_available_dates():
            print "Collecting historical data for %s..." % \
                  selected_date
            self.calculate_and_save_sub_indexes(
                selected_date, months, r)

        print "Completed."

    def calculate_and_save_sub_indexes(self, selected_date,
                                       months_fwd, r):
        current_dt = self.webpage.get_date_from_web_date(
            selected_date)

        for i in range(1, months_fwd+1):
            # Get settlement date of the expiring month
            expiry_dt = self.utility.fwd_expiry_date(
                current_dt, i)
```

```
            # Get calls and puts of expiring month
            dataset, update_dt = self.get_data(current_dt,
                                               expiry_dt)
            if not dataset.empty:
                sub_index = self.calculator.calculate_sub_index(
                    dataset, update_dt, expiry_dt, r)
                self.save_vstoxx_sub_index_to_csv(
                    current_dt, sub_index, i)

    def save_vstoxx_sub_index_to_csv(self, current_dt,
                                     sub_index, month):
        subindex_df = None
        try:
            subindex_df = pd.read_csv(self.sub_index_store_path,
                                      index_col=[0])
        except:
            subindex_df = pd.DataFrame()

        display_date = current_dt.strftime(self.csv_date_format)
        subindex_df.set_value(display_date,
                              "I" + str(month),
                              sub_index)
        subindex_df.to_csv(self.sub_index_store_path)

    def get_data(self, current_dt, expiry_dt):
        """ Fetch and join calls and puts option series data """
        calls, dt1 = self.webpage.get_option_series_data(
            True, current_dt, expiry_dt)
        puts, dt2 = self.webpage.get_option_series_data(
            False, current_dt, expiry_dt)
        option_series = calls.join(puts, how='inner')
        if dt1 != dt2:
            print "Error: 2 different underlying prices."

        return option_series, dt1
```

Analyzing the results

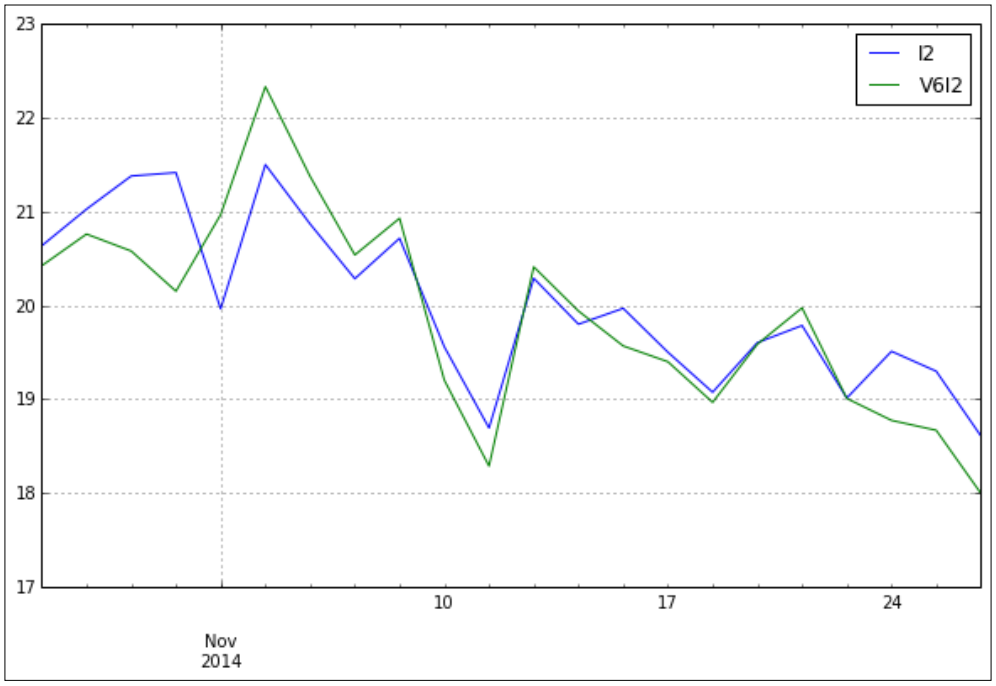
In the `data` folder of our working directory, we should have the following two files by now: `vstoxx.csv` and `vstoxx_sub_indexes.csv`. The sub-indexes file contains the computed values of the sub-index. Using the following Python code, we can plot the values of 2 months to expiry sub-index values:

```
import pandas as pd

vstoxx_sub_indexes = pd.read_csv('data/vstoxx_sub_indexes.csv',
```

```
                                    index_col=[0],
                                    parse_dates=True, dayfirst=False)
vstoxx = pd.read_csv('data/vstoxx.csv', index_col=[0],
                     parse_dates=True, dayfirst=False)

start_dt = min(vstoxx_sub_indexes.index.values)
vstoxx = vstoxx[vstoxx.index >= start_dt]

from pylab import *
new_pd = pd.DataFrame(vstoxx_sub_indexes["I2"])
new_pd = new_pd.join(vstoxx["V6I2"], how='inner')
new_pd.plot(figsize=(10, 6), grid=True)
```

Let's compare these values of the 2 months to expiry option values to see how well our model performs:

```
>>> show()
Populating the interactive namespace from numpy and matplotlib
<matplotlib.axes.AxesSubplot at 0x10f090c10>
```



We can see that the calculated values tend to move in the same direction as the actual V6I2 values.

Calculating the VSTOXX main index

From the `vstoxx.csv` file saved earlier, we can use these values to calculate the VSTOXX index values. The formula to calculate the index is given as follows:

$$VSTOXX_i = \sqrt{\left[\frac{N_i}{N_{365}}\left(\frac{N_{i+1}-N_{30}}{N_{i+1}-N_i}\right)VSTOXX_1^2 + \frac{N_{i+1}}{N_{365}}\left(\frac{N_{30}-N_i}{N_{i+1}-N_i}\right)VSTOXX_2^2\right]\frac{365}{30}}$$

Here,

N_{365} and N_{30} is the number of seconds in one year and 30 days respectively.

N_i is the time left to the expiry of the nearest OESX in seconds.

N_{i+1} is the time left to the expiry of the second nearest OESX in seconds.

The implementation of the preceding formula is given in the `calculate_vstoxx_index` method:

```
import math

def calculate_vstoxx_index(dataframe, col_name):
    secs_per_day = float(60*60*24)
    utility = OptionUtility()

    for row_date, row in dataframe.iterrows():
        # Set each expiry date with an
        # expiration time of 5p.m
        date = row_date.replace(hour=17)

        # Ensure dates and sigmas are in legal range
        expiry_date_1 = utility.get_settlement_date(date)
        expiry_date_2 = utility.fwd_expiry_date(date, 1)
        days_diff = (expiry_date_1-date).days
        sigma_1, sigma_2 = row["V6I1"], row["V6I2"]
        if -1 <= days_diff <= 1:
            sigma_1, sigma_2 = row["V6I2"], row["V6I3"]
        if days_diff <= 1:
            expiry_date_1 = expiry_date_2
            expiry_date_2 = utility.fwd_expiry_date(date, 2)

        # Get expiration times in terms of seconds
        Nti = (expiry_date_1-date).total_seconds()
```

```
            Nti1 = (expiry_date_2-date).total_seconds()

            # Calculate index as per VSTOXX formula in seconds
            first_term = \
                (Nti1-30*secs_per_day)/ \
                (Nti1-Nti)*(sigma_1**2)*Nti/ \
                (secs_per_day*365)
            second_term = \
                (30*secs_per_day-Nti)/ \
                (Nti1-Nti)*(sigma_2**2)*Nti1/ \
                (secs_per_day*365)
            sub_index = math.sqrt(365.*(first_term+second_term)/30.)
            dataframe.set_value(row_date, col_name, sub_index)

        return dataframe
```

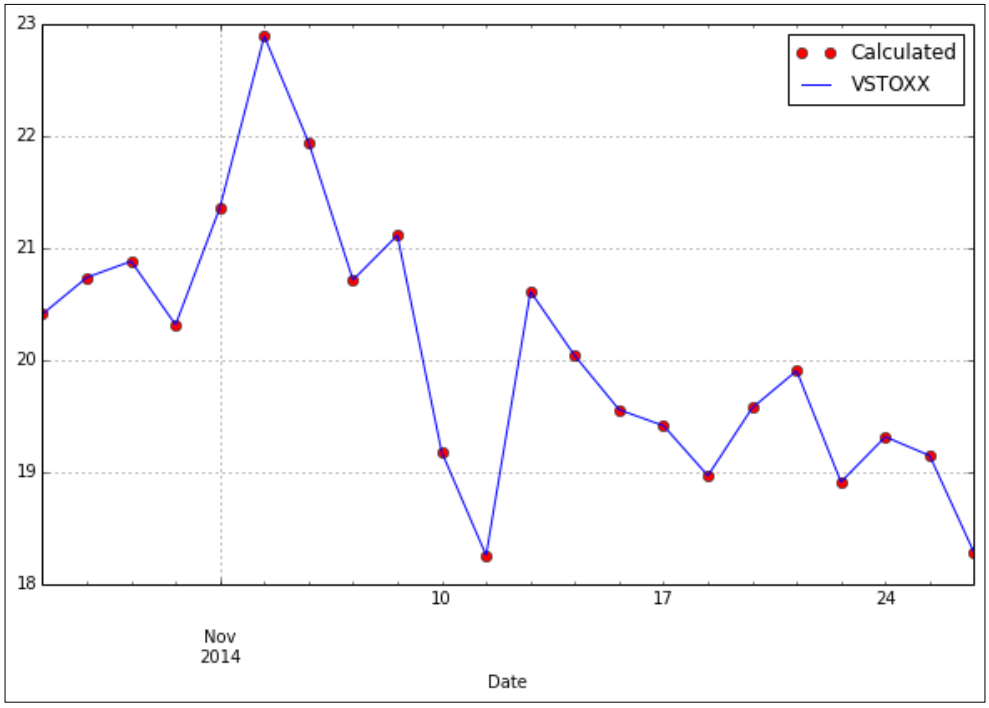
Note that the function requires the use of the `OptionUtility` class discussed earlier to calculate the settlement date. Adjustments are made if the calculation date is one or less than one day to expiry. If the calculation date falls beyond the expiry date of the same month, the values of the following month are used.

To see how our calculated values can be compared to the VSTOXX actual values, we can take a sample from the VSTOXX data file and feed it into our `calculate_vstoxx_index` function. The following Python code demonstrates this. Since the data file dates a long time back, we will just take a sample of the last 100 sub-index values:

```
>>> sample = vstoxx.tail(100)  # From the previous section
>>> sample = calculate_vstoxx_index(sample, "Calculated")
>>>
>>> vstoxx_df = sample["V2TX"]
>>> calculated_df = sample["Calculated"]
>>> df = pd.DataFrame({'VSTOXX' : sample["V2TX"],
...                'Calculated' : sample["Calculated"]})
>>>  df.plot(figsize=(10, 6), grid=True, style=['ro','b'])
```

This gives us the following output:

```
<matplotlib.axes.AxesSubplot at 0x10c971f90>
```



The calculated values in red dots appear to be very close to the actual VSTOXX values.

Summary

In this chapter, we looked at volatility derivatives and their uses by investors to diversify and hedge their risk in equity and credit portfolios. Since long-term investors in equity funds are exposed to downside risk, volatility can be used as a hedge for the tail risk and in replacement for the put options. In the United States, the CBOE Volatility Index (VIX) measures the short-term volatility implied by S&P 500 stock index option prices. In Europe, the VSTOXX market index is based on the market prices of a basket of OESX and measures the implied market volatility over the next 30 days on the EURO STOXX 50 Index. Many people around the world use the VIX as a popular measurement tool for the stock market volatility over the next 30-day period. To help us better understand how the VSTOXX market index is calculated, we looked at its components and at formulas used in determining its value.

The STOXX and Eurex Exchange websites provide the historical daily data of the main index and its sub-indexes. To help us determine the relationship between the EURO STOXX 50 Index and VSTOXX, we downloaded this data with Python, merged them, and performed a variety of financial analytics. We came to the conclusion that they are negatively correlated. This relationship presents a viable way of avoiding frequent rebalancing costs by trading strategies based on benchmarking. The statistical nature of volatility allows volatility derivative traders to generate returns by utilizing mean-reverting strategies, dispersion trading, and volatility spread trading, among others.

The VSTOXX consists of eight sub-indexes that represent the calculated volatility index from the EURO STOXX 50 Index options expiring in 1, 2, 3, 6, 9, 12, 18, and 24 months. Since the VSTOXX index represents the volatility outlook for the next 30 days, we gathered the OESX call and put prices from the Eurex website and calculated the sub-indexes for the 2 month forward expiry date.

Finally, we studied the component weighing formula of the VSTOXX sub-indexes and used Python to calculate the VSTOXX main index value to give us an estimate of the volatility outlook for the next 30 days.

In the next chapter, we will take a look at managing big data in finance with Python.

7
Big Data with Python

With the advent of cloud computing technologies, big data has become increasingly commonplace. What is big data exactly and how can you work with big data to gather useful information? How different is big data from the kind of data we come across everyday? This chapter will specifically answer these questions and introduce you to the use of big data in finance. Big data tools provide the scalability and reliability of analyzing large volumes of data coming from multiple sources. In meeting these big data needs, Apache Hadoop became the primary choice for financial institutions and enterprises. As such, it is crucial for financial engineers to be familiar with Hadoop for financial applications.

As we begin to process large datasets, we also need to find an avenue to store this data. The de facto standard for relational database management was **Structured Query Language** (**SQL**). The nature of digital data is varied, and other means of storing data became the motivation for non-SQL products. One such nonrelational database mechanism is **NoSQL**, which stands for **Not Only SQL**. Besides being able to use SQL-like language for data management, NoSQL allows the storage of nonstructured data, such as key values, graphs, or documents. Because of its simplicity in design, it can also be said to be faster in certain circumstances. One area where NoSQL is used in finance is the storage of incoming tick data. This chapter will introduce you to the use of NoSQL for tick data storage.

In this chapter, we will cover the following topics:

- An introduction to big data, Apache Hadoop, and its components
- Getting Hadoop and running a QuickStart virtual machine
- Using the Hadoop HDFS file store

- Performing a simple word count on an e-book using MapReduce with Python
- Testing the MapReduce program before running it on Hadoop
- Performing a MapReduce operation on the daily price changes of a stock
- Analyzing the results of the MapReduce operation with Python
- An introduction to NoSQL
- Getting and running MongoDB
- Getting and installing the PyMongo module for Python
- An introduction to databases and collections with PyMongo
- Insert, delete, find, and sort tick data with a NoSQL collection

Introducing big data

There has been a lot of excitement about big data and the kind of skills involved with it. Before beginning this chapter, it is important to define what big data is and how you can work with big data to gather useful information. How different is big data from the kind of data we come across everyday, say news stories, reports, literature, or even audio?

Big data is actually data captured at a high velocity, and it accumulates in such large quantities that it takes up terabytes or petabytes of storage. Common software tools are inadequate for capturing, processing, maintaining, and managing such data with a short period of tolerance. Analytical tools are applied on these large datasets to uncover the information and relationships that could possibly be used for forecasting or other analytical activities.

With the advent of cloud computing technologies, big data has become increasingly commonplace. Massive amounts of information can be stored in the cloud at lower costs. The move from relational databases to nonrelational solutions, such as NoSQL, allows nonstructured data to be captured at a more rapid rate. By performing data analytics on the captured information, companies are able to improve their operational efficiency, analyze patterns, run targeted marketing campaigns, and improve customer satisfaction.

Financial sector companies are integrating big data analytics into their operations. Analytics are performed in real time on customer transactions to identify abnormal behavior and detect fraud. Customer records, spending habits, and even activities on social media sites can be used to promote products and services by customer segregation. Big data tools provide the scalability and reliability of analyzing big data in the area of risk and credit analytics, with the data coming in from multiple sources.

Hadoop for big data

Apache Hadoop is a 100 percent open source software framework used for two important fundamental tasks: storing and processing big data. It has been the leading big data tool for distributed parallel processing of data stored across multiple servers and is able to scale without limits. Because of its scalability, flexibility, fault tolerance, and low-cost features, many cloud-based solution vendors, financial institutions, and enterprises use Hadoop for their big data needs.

The Hadoop framework contains modules that are critical to its functions: the **Hadoop Distributed File System** (**HDFS**), **Yet Another Resource Negotiator** (**YARN**), and **MapReduce** (**MapR**).

HDFS

HDFS is a file system unique to Hadoop that is designed to be scalable and portable, and allows large amounts of file storage over multiple nodes in a Hadoop cluster spanning gigabytes or terabytes of data. Data in a cluster is split into smaller blocks of 128 Megabytes typically and distributed throughout the cluster. The MapReduce data processing functions are performed on smaller subsets of the larger datasets, thereby providing the scalability needed for big data processing. The HDFS file system uses TCP/IP socket communications to serve data over the network as one big file system.

YARN

YARN is a resource management and scheduling platform that manages the CPU, memory, and storage for applications running on a Hadoop cluster. It contains the components responsible for: allocating resources among applications running within the same cluster while obeying constraints, such as queue capacities and user limits; scheduling tasks based on the resource requirements of each application; negotiating appropriate resources from the scheduler; and tracking and monitoring progress of the running applications and their resource usage.

MapReduce

MapReduce is the software programming paradigm of Hadoop used for processing and generating large datasets. For a developer, MapReduce is the probably the most important programming component in Hadoop. It is made up of two functions: map and reduce. The **map** function processes a key-value pair to generate an intermediate key-value pair. The **reduce** function merges all intermediate values with the same intermediate key and produces a result. MapReduce eliminates the need for moving data over the network to be processed by the software, and instead brings the processing software to the data.

MapReduce uses Java predominantly. Other languages, such as SQL and Python, can be implemented for MapReduce using the Hadoop streaming utility.

Is big data for me?

Perhaps you might be working on a file that is hundreds of megabytes in size, and your current data analysis tool is performing too slowly. Since big data typically involves terabytes or gigabytes worth of storage, you might be wondering, Is Apache Hadoop for me?

Note that Hadoop implements one general computation by mapping every single entity on your data, and then performing some reduction computation to add up the individual parts. You might be able to achieve the same grouping and counting function using programming languages, such as SQL and Python. In addition, writing the code allows you to express the computational flow more easily.

Or you might want to consider migrating your data analysis tools to pandas or R. They are very powerful and able to handle gigabytes of data when coded efficiently with no memory leaks. Most commercial SQL servers are up to the task. Besides, memory and storage costs are so affordable that you could be performing heavy computations on your local workstation.

If your data storage runs into terabytes, then you are out of luck. Without many choices left, Apache Hadoop and other alternative big data analysis tools such as Apache Spark seem to be your best hope for scalability, affordability, and fault-tolerance.

Getting Apache Hadoop

The official page for Apache Hadoop is `http://hadoop.apache.org`. Here, you can find in-depth documentation, manuals, and releases of Apache Hadoop. Hadoop is written in Java and requires JVM installed on your single-node setup to run. It is supported on both GNU/Linux and Windows.

Since the purpose of this chapter is to get introduced to Python programming for Apache Hadoop, a quick way to get our hands on a complete Hadoop ecosystem would be most ideal. Cloud vendor Cloudera hosts a number of free QuickStart VMs that contain a single-node Apache Hadoop cluster, complete with sample scripts and ready links to help us dive straight into managing our cluster. The following sections describe how to get a Hadoop VM running on your machine.

Getting a QuickStart VM from Cloudera

The download link to Hadoop QuickStart VMs from Cloudera is `http://www.cloudera.com/content/support/en/downloads/quickstart_vms.html`. The VM image comes installed with the CentOS 6.4 Linux operating system and is available for VMWare, VirtualBox, and KVM virtual machine platforms. The version of the QuickStart VM that we will use is *Quick Start VM with CDH 5.3.x*. Let's choose the free and open source VirtualBox as our VM interface.

Since the VMs are 64-bit, they require a 64-bit host OS, and a virtualization platform that can support a 64-bit guest OS. The file size is 3 GB, and it requires 4 GB of RAM in the virtual machine.

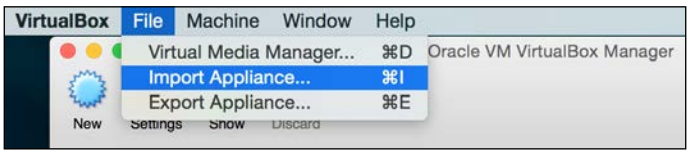
Getting VirtualBox

VirtualBox runs on Windows, Linux, Macintosh, and Solaris hosts and supports a large number of guest operating systems, including but not limited to OpenSolaris, OS/2, and OpenBSD. The link to get VirtualBox is `https://www.virtualbox.org/wiki/Downloads`.

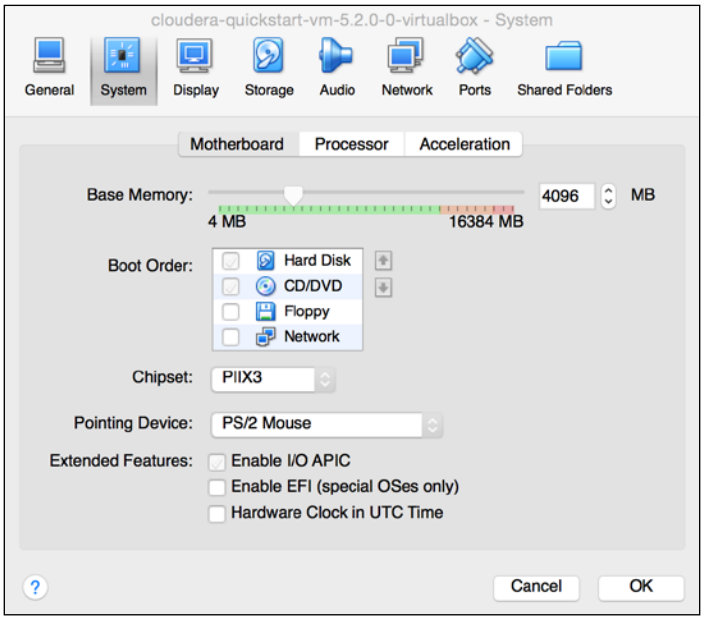
Running Cloudera VM on VirtualBox

The following steps describe how to get Cloudera's Hadoop VM running smoothly on VirtualBox:

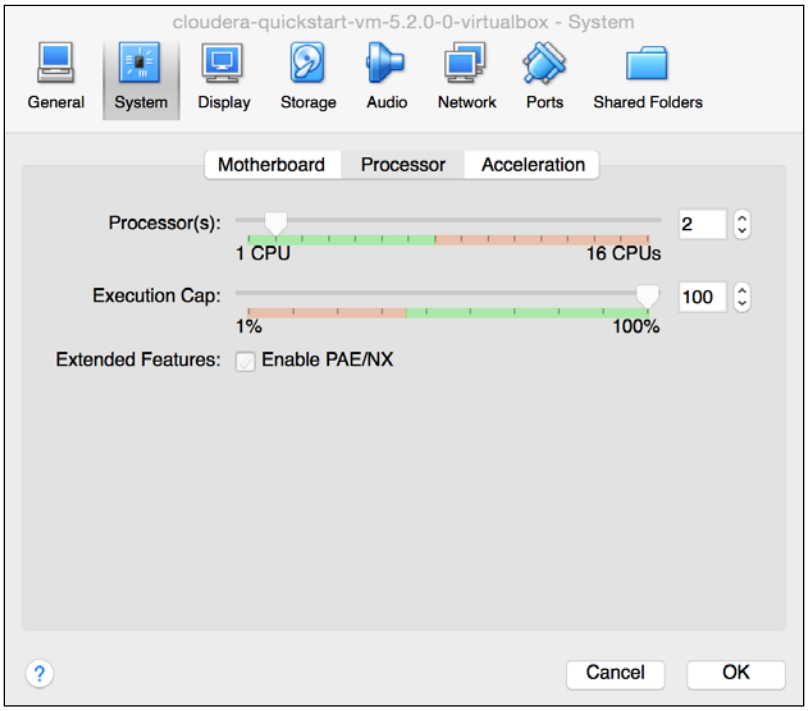
1. Unzip the download package from Cloudera to a folder of your choice.
2. Open VirtualBox. From the menu bar, go to **File**, then select **Import Appliance**. Follow the steps to select the unzipped virtual machine from step 1. This will add the Cloudera VM image to the list of machines; it is compatible to run on VirtualBox:



3. Select the Cloudera QuickStart machine from the virtual machine list. Click on **Settings**. Go to **System** tab, and then click on the **Motherboard** tab. Ensure that you have at least 4096 MB of RAM selected:

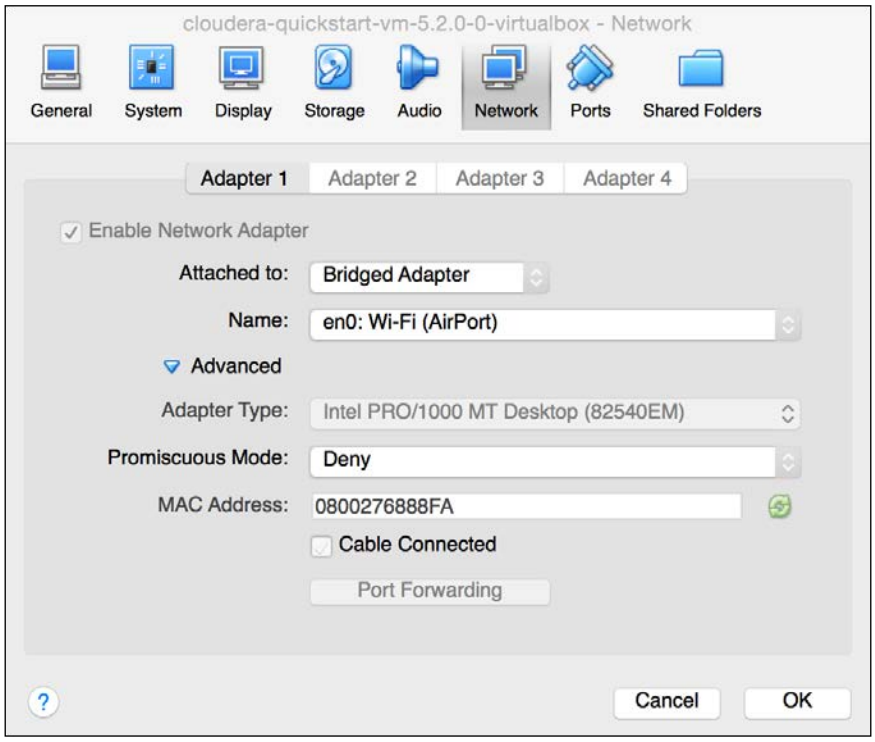


4. Go to the **Processor** tab. Ensure that you have at least two **Processor(s)** selected:



5. Go to the **Acceleration** tab, and ensure that all **Hardware Virtualization** checkboxes are selected.

6. Go to **Network** options and ensure that **Bridged Adapter** is selected:



7. Click on **OK** to save the changes.

8. On the same virtual machine, click on **Start**. This will start the CentOS virtual machine. It might take a few minutes to boot into the operating system:

With CentOS now running, we are automatically logged in as the `cloudera` user. We will do all the computations and scripting in this virtual machine. If required, the username and password credentials are both `cloudera`. This includes the sudo privileges for the root account, root MySQL, Hue, and Cloudera manager. The home directory is `/home/cloudera/`.

A word count program in Hadoop

Perhaps the simplest way to get started with understanding programming for Hadoop is a simple word count functionality on a fairly large electronic book. The map program will read in every line of the text separated by a space or tab and return a key-value pair, which is by default assigned to a count of 1. The reduce program will read in all key-value pairs from the map program and sum up the number of similar words. Hadoop will produce an output file that contains a list of words in the book and the number of times the words have appeared.

Downloading sample data

Project Gutenberg hosts over 100,000 free e-books in HTML, EPUB, Kindle, and plain-text UTF-8 formats. For our testing with a sample e-book, let's use *Ulysses by James Joyce*. The link for the plain text UTF-8 file is `http://www.gutenberg.org/ebooks/4300.txt.utf-8`. Using Firefox or any other web browser available in the CentOS virtual machine, you can download the file from the URL, and save it as `pg4300.txt` in the `Downloads` folder of our home directory, at `/home/cloudera/Downloads`.

Once we have our target e-book downloaded onto our local drive, it is time to copy the e-book data in the Hadoop HDFS file store for processing. In the Terminal, run the following command:

```
[cloudera@quickstart ~]$ hadoop fs -copyFromLocal
/home/cloudera/Downloads/pg4300.txt pg4300.txt
```

This will copy the e-book to the Hadoop HDFS store with the same filename `pg4300.txt`.

To ensure that our copy operation is successful, use the `hadoop fs -ls` command to give us the following output:

```
[cloudera@quickstart ~]$ hadoop fs -ls
Found 1 items -rw-r--r--   1 cloudera cloudera    1573150 2014-12-05
22:26 pg4300.txt
```

The Hadoop file store shows that our file has been copied successfully.

The map program

We will run the map program in Python on Hadoop. Insert the following code in a file named `mapper.py` so that we can use this later:

```
#!/usr/bin/python
import sys

for line in sys.stdin:
    for word in line.strip().split():
        print "%s\t%d" % (word, 1)
```

When `mapper.py` is executed, the interpreter will read in the input buffer that consists of text. All the text will be broken down by empty whitespace characters, and each word will be assigned to a count of 1, separated by a tab character.

A user-friendly text editor in CentOS is gedit. Create a new folder named `word_count` in your home directory and save the file in `/home/cloudera/word_count/mapper.py`.

The `mapper.py` file needs to be recognized as an executable file in Linux. From the Terminal, run the following command:

```
chmod +x /home/cloudera/word_count/mapper.py
```

This will ensure that our map program can run without restrictions.

The reduce program

To create our reduce program, paste the following Python code into a text file named `reduce.py`, and place it in the same folder at `/home/cloudera/word_count/` so that we can use this later:

```
#!/usr/bin/python
import sys

current_word = None
current_count = 1

for line in sys.stdin:
    word, count = line.strip().split('\t')
```

```
        if current_word:
            if word == current_word:
                current_count += int(count)
            else:
                print "%s\t%d" % (current_word, current_count)
                current_count = 1

        current_word = word

    if current_count > 1:
        print "%s\t%d" % (current_word, current_count)
```

The reduce program will read in all the key-value pairs from the map program. The current word and word count from the key-value pair is obtained by stripping off the tab character and comparing it with the previous occurrence of the word. Every similar occurrence increases the word count by one with the end result: printing the word itself and its count separated by the tab character.

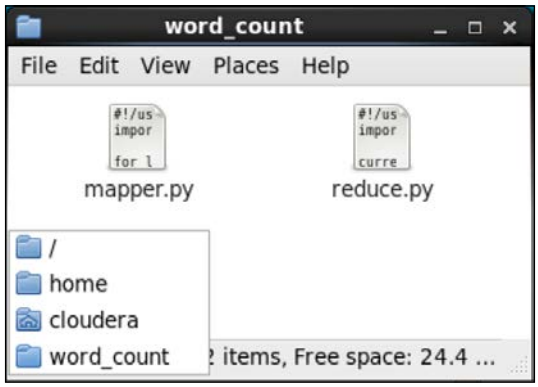
Again, the `reduce.py` file needs to be recognized as an executable file in Linux. From the Terminal, run the following command:

```
chmod +x /home/cloudera/word_count/reduce.py
```

This will ensure that our reduce program can run without restrictions.

Testing our scripts

Before running our map and reduce program on Hadoop, we can run the scripts locally on our machine first to make sure everything works as intended. If we navigate to our `word_count` folder, we should see the following files:



In the Terminal, run the following command on our `mapper.py` Python file:

```
[cloudera@quickstart ~]$ echo "foo foo quux labs foo bar quux" |
/home/cloudera/word_count/mapper.py
```

This will produce the following output:

```
foo    1
foo    1
quux    1
labs    1
foo    1
bar    1
quux    1
```

As expected, our map program will issue a count of one to every word read into its input buffer, and print the results on each line.

In the Terminal, run the following command on our `reduce.py` Python file:

```
[cloudera@quickstart ~]$ echo "foo foo quux labs foo bar quux" |
/home/cloudera/word_count/mapper.py | sort -k1,1 |
/home/cloudera/word_count/reduce.py
```

This will produce the following output:

```
bar    1
foo    3
labs    1
quux    2
```

Our reduce program sums up the number of similar words encountered by the map program, prints the results on each line, and sorts them in ascending order. It works as intended.

Running MapReduce on Hadoop

We are now ready to run our MapReduce operation on Hadoop. In the Terminal, type the following command:

```
hadoop jar \
/usr/lib/hadoop-0.20-mapreduce/contrib/streaming/hadoop-streaming-
2.5.0-mr1-cdh5.3.0.jar \
-file /home/cloudera/word_count/mapper.py \
```

```
-mapper /home/cloudera/word_count/mapper.py \
-file /home/cloudera/word_count/reduce.py \
-reducer /home/cloudera/word_count/reduce.py \
-input pg4300.txt \
-output pg4300-output
```

We will use the Hadoop Streaming utility to enable the use of Python scripts as the map and reduce operation. The Java JAR file for the Hadoop Streaming operation is assumed to be `hadoop-streaming-2.5.0-mr1-cdh5.3.0.jar` and is located in the folder at `/usr/lib/hadoop-0.20-mapreduce/contrib/streaming/`. On other systems, the Hadoop Streaming utility JAR file may be placed in a different folder or may use a different filename. The provided additional arguments will specify our input files, and the output directory that is required by the Hadoop operation.

Once the Hadoop operation starts, we should get an output similar to the following:

```
14/12/06 09:39:37 INFO client.RMProxy: Connecting to ResourceManager
at /0.0.0.0:8032
14/12/06 09:39:38 INFO client.RMProxy: Connecting to ResourceManager
at /0.0.0.0:8032
14/12/06 09:39:38 INFO mapred.FileInputFormat: Total input paths to
process : 1
14/12/06 09:39:38 INFO mapreduce.JobSubmitter: number of splits:2
14/12/06 09:39:39 INFO mapreduce.JobSubmitter: Submitting tokens for
job: job_1417846146061_0002
14/12/06 09:39:39 INFO impl.YarnClientImpl: Submitted application
application_1417846146061_0002
14/12/06 09:39:39 INFO mapreduce.Job: The url to track the job:
http://quickstart.cloudera:8088/proxy/application_1417846146061_0002/
14/12/06 09:39:39 INFO mapreduce.Job: Running job:
job_1417846146061_0002
14/12/06 09:39:46 INFO mapreduce.Job: Job job_1417846146061_0002
running in uber mode : false
14/12/06 09:39:46 INFO mapreduce.Job:  map 0% reduce 0%
14/12/06 09:39:53 INFO mapreduce.Job:  map 50% reduce 0%
```

A number of interesting things begin to happen in Hadoop. It tells us that the MapReduce operation can be tracked at `http://quickstart.cloudera:8088/proxy/application_1417846146061_0002/`. It also tells us about the progress of the operation, which is currently at 50 percent, since it could take some time to process the whole book.

When the operation is completed, the last few lines of the output contain more information of interest:

```
File Input Format Counters
Bytes Read=1577103
File Output Format Counters
Bytes Written=527716
14/12/06 09:40:03 INFO streaming.StreamJob: Output directory: pg4300-
output
```

It tells us that the MapReduce operation has completed successfully, and the results are written to our target output folder at `pg4300-output`. To verify this, we can take a peek at our HDFS file store:

```
[cloudera@quickstart ~]$ hadoop fs -ls
Found 2 items
drwxr-xr-x   - cloudera cloudera          0 2014-12-06 09:40 pg4300-
output -
rw-r--r--   1 cloudera cloudera    1573150 2014-12-05 22:26
pg4300.txt
```

We can see that a new folder has been added to our file store, which is one of our target output folders. Let's take a peek at the contents of this output folder:

```
[cloudera@quickstart ~]$ hadoop fs -ls pg4300-output
Found 2 items
-rw-r--r--   1 cloudera cloudera          0 2014-12-06 09:40 pg4300-
output/_SUCCESS


-rw-r--r--   1 cloudera cloudera     527716 2014-12-06 09:40 pg4300-
output/part-00000
```

Here, we can see that Hadoop churns out two files in our target folder. The `_SUCCESS` file is just an empty file that tells us that the MapReduce operation by Hadoop is successful. The second file, `part-00000`, contains the results of the reduce operation. We can inspect the output file with the `fs -cat` command:

```
[cloudera@quickstart ~]$ hadoop fs -cat pg4300-output/part-00000
"Come  1
"Defects,"1
"I1
```

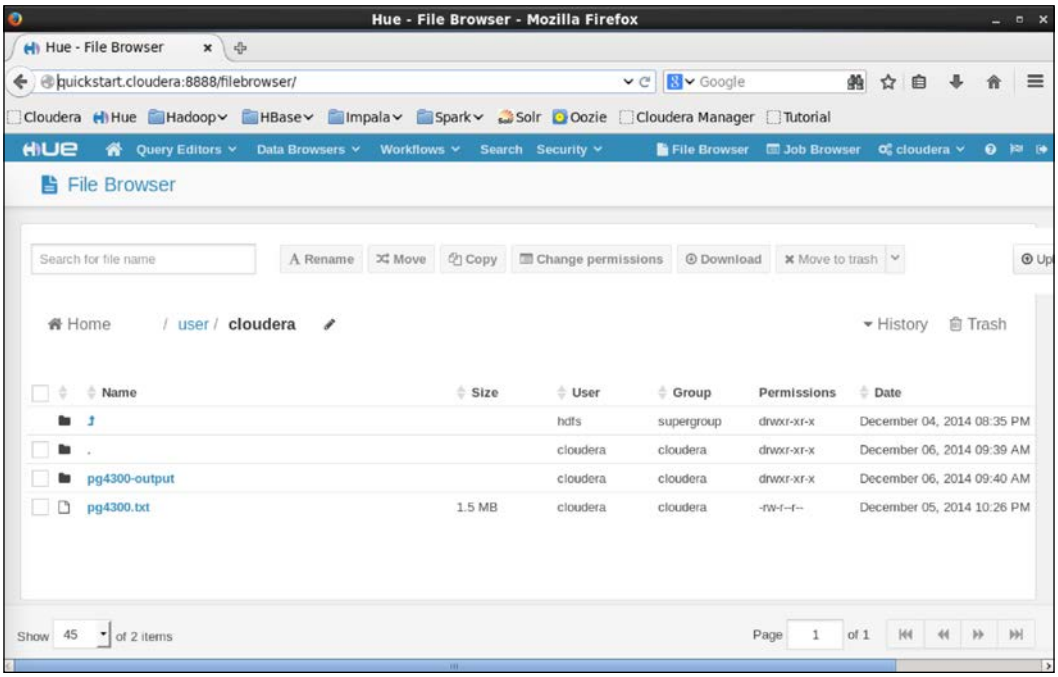
```
"Information 1
"J"1
"Plain 2
"Project5
"Right 1
"Viator"1
#4300] 1
```

The list of words is too long to be printed, but you get the idea.

Hue for browsing HDFS

Besides using the Terminal to navigate to the HDFS file store, another way of having a GUI interface to browse the HDFS file store for troubleshooting or invoking any other regular file operation is through the Hue web interface manager. Hue is also ideal for navigating large amounts of output information, such as viewing the full output of word counts.

Hue can be accessed from any web browser through the URL `http://quickstart.cloudera:8888/filebrowser/`.



The link to Hue is included in the quick links bookmark of Firefox. The **File Browser** hyperlink is located in the top-right section of the Hue web page.

Going deeper – Hadoop for finance

Now that we know how to use Hadoop to perform a simple word count on a fairly large text file, we can take a step further and use Hadoop for quantitative analysis. For a start, we can count the number of historical intraday percentage price changes of a stock.

Obtaining IBM stock prices from Yahoo! Finance

To obtain a dataset, we can use the historical stock prices available from Yahoo! Finance. Using Firefox or any web browser in your CentOS environment, you can download the historical daily prices for a stock counter as a CSV file using the following link

```
http://ichart.finance.yahoo.com/table.csv?s=IBM
```

In this example, we will use IBM as our example stock. Download the file to the `Downloads` folder of your home directory and rename it as `ibm.csv`. If we take a look at the contents of the CSV file, the daily stock prices go all the way back to 1962.

Then run the following command in the Terminal to copy our target CSV file to the Hadoop HDFS file store:

```
[cloudera@quickstart /]$ hadoop fs -copyFromLocal
/home/cloudera/Downloads/ibm.csv ibm.csv
```

If we do a directory listing on the HDFS, we can see that our CSV file has been copied successfully:

```
[cloudera@quickstart /]$ hadoop fs -ls
Found 4 items
drwxr-xr-x   - cloudera cloudera          0 2014-12-06 15:01 .Trash
-rw-r--r--   1 cloudera cloudera     685955 2014-12-06 14:53 ibm.csv
drwxr-xr-x   - cloudera cloudera          0 2014-12-06 09:40 pg4300-
output
-rw-r--r--   1 cloudera cloudera    1573150 2014-12-05 22:26
pg4300.txt
```

Modifying the map program

Now all that we need to do is to create our map and reduce program to study the stock prices.

In our home directory, create a new folder called `stock`. We can copy the `mapper.py` and `reduce.py` files used in the previous section to this folder at `/home/cloudera/stock/`.

Open `mapper.py` with gedit or any other text editor to begin editing our map program. We will use the following Python code:

```
#!/usr/bin/python
import sys

is_first_line = True
for line in sys.stdin:
   if is_first_line:
      is_first_line = False
      continue

   row =  line.split(',')
   open_price = float(row[1])
   close_price = float(row[-3])
   change = (open_price-close_price)/open_price * 100
   change_text = str(round(change,1)) + "%"
   print "%s\t%d" % (change_text, 1)
```

This will tell our map program to ignore the first line upon reading the CSV file since the top row contains redundant header information. The following rows contain the date, open, high, low, close, volume, and adjusted close prices separated by comma characters. We are interested in using the open and close prices to calculate the percentage change in the stock prices for the day. The program outputs the value to one decimal place with a default count of one, separated by a tab character.

We will reuse the same reduce program without any modifications. Ensure that the two files have the required permissions to run by issuing the following commands in the Terminal:

```
chmod +x /home/cloudera/stock/mapper.py
chmod +x /home/cloudera/stock/reduce.py
```

Testing our map program with IBM stock prices

Before running our program in Hadoop, let's test our map program with the CSV data file we earlier downloaded with the following Unix command:

```
[cloudera@quickstart /]$ cat /home/cloudera/Downloads/ibm.csv |
/home/cloudera/stock/mapper.py
```

This will give us a long list of the daily price changes in percentage form. The following output is a snippet of the last few values of the output by the Terminal:

```
0.0%  1
-0.7%  1
1.8%  1
1.8%  1
1.0%  1
-0.9%  1
1.1%  1
```

It seems like our map program output is very similar to the earlier word count map program.

Running MapReduce to count intraday price changes

Our map and reduce program is ready to run on Hadoop. The Unix command is similar to the one we used in the word count program, albeit with a few changes:

```
hadoop jar \
/usr/lib/hadoop-0.20-mapreduce/contrib/streaming/hadoop-streaming-
2.5.0-mr1-cdh5.3.0.jar \
-file /home/cloudera/stock/mapper.py \
-mapper /home/cloudera/stock/mapper.py \
-file /home/cloudera/stock/reduce.py \
-reducer /home/cloudera/stock/reduce.py \
-input ibm.csv \
-output stock-output
```

The Hadoop MapReduce operation will start running. When this has finished, we should get an output similar to the last few lines as follows:

```
Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
File Input Format Counters
Bytes Read=687042 File Output Format Counters
Bytes Written=1211
14/12/06 16:46:00 INFO streaming.StreamJob: Output directory: stock-
output
```

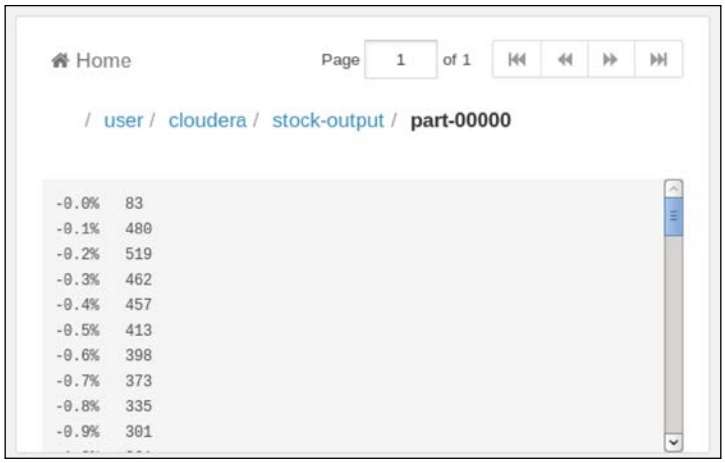
As expected, the output results will be stored in the `stock-output` folder of the HDFS file store.

Before beginning to study the result for analysis, we need to copy the file from the HDFS file store to our local working folder. Run the following command in the Terminal:

```
[cloudera@quickstart /]$ hadoop fs -copyToLocal  stock-output/part-
00000 /home/cloudera/stock/
```

The output file can be accessed from `/home/cloudera/stock/part-000000` on our local drive.

Using Hue, we can view the output results as well:



Performing analysis on our MapReduce results

Before we can display the graphs using Python in the CentOS environment, the `matplotlib` module needs to be installed. This is fairly easy. From the Terminal, run the following command:

```
sudo yum install python-matplotlib
```

It will take some time to download and install the required modules. Press `y` and the *Enter* key to install when prompted. Once completed, the `matplotlib` module is loaded into our Python environment.

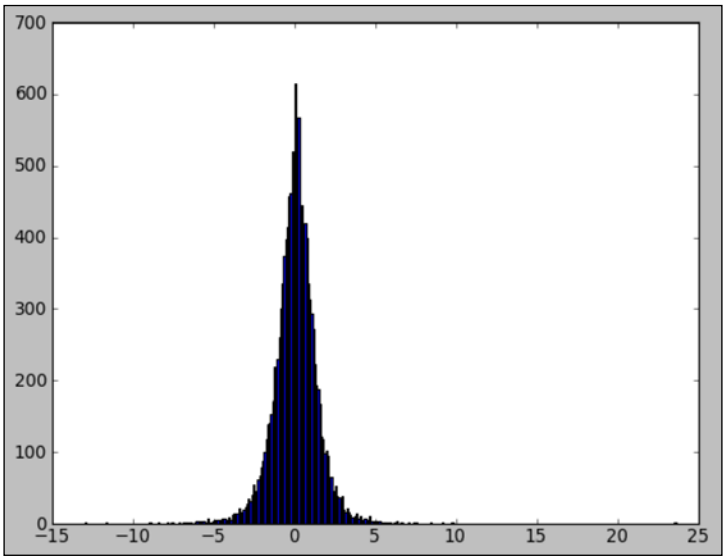
The following code demonstrates a simple analysis by displaying the results onto a bar chart:

```
import matplotlib.pyplot as plt

with open('/home/cloudera/stock/part-00000', 'rb') as f:
   x, y = [], []
   for line in f.readlines():
      data = line.split()
      x.append(float(data[0].strip('%')))
      y.append(float(data[1]))
    print "Max: ", max(x)
    print "Min: ", min(x)
    plt.bar(x, y, width=0.1)
    plt.show()
```

We will read each line of the output file and separate the results into lists of the `x` and `y` values, which are then fed into the `bar` function to be plotted as a graph using the `matplotlib.pyplot` module. The `width` parameter of the `bar` function helps you narrow down the width of the bar to a small value of 0.1 since we have too many bars to be displayed on the chart. We are also interested in the maximum and minimum values that the daily IBM stock price changes can get.

Save the Python code to a file, say `analysis.py` to the folder at `/home/cloudera/stock/`. In the Terminal, run this Python file:

```
[cloudera@quickstart /]$ python /home/cloudera/stock/analysis.py
Max:  23.5
Min:  -13.0
```



We concluded that the maximum intraday price change for IBM was 23.5 percent at one time and a minimum of -13.0 percent. The intraday price change of IBM is distributed around the mean of 0.0 percent.

Introducing NoSQL

Many cloud solution vendors offer storage facilities similar to the use of NoSQL by storing unstructured data in a document type of model. In this section, we will explore NoSQL as a means of storing financial data. There are also a number of open source databases available for free that support NoSQL.

Getting MongoDB

MongoDB is a free and open source document database written in C++. The official site for MongoDB is `http://www.mongodb.org`. MongoDB is available for Linux, Windows, Mac OS X, and Solaris. Head to `http://www.mongodb.org/downloads` to download and install MongoDB on your local workstation. MongoDB management services are typically added to your operating system's runtime environment when installation is complete and run from the command line. The documentation also includes instructions to get your `mongod` service started and running on your machine.

Creating the data directory and running MongoDB

Before starting MongoDB for the first time, a directory is needed where the `mongod` process will write data. The `mongod` process uses `/data/db` as the default directory.

Let's create the `data` directory in a folder of our choice. Ensure that the user account has the proper read and write privileges.

Running MongoDB from Windows

In Windows, open **Command Prompt**, and navigate to your working directory with the `cd` command. Then, create a folder with the following command:

```
$ md data\db
```

To run the `mongod` service, enter the following command:

```
$ c:\mongodb\bin\mongod.exe -dbpath c:\test\data\db
```

Here, it is assumed that the MongoDB installation files are located at `c:\mongo`, and our working directory is `c:\test`. Depending on how you specify the MongoDB installation, these paths could differ from yours.

Running MongoDB from Mac OS X

If you are using Mac OS X, in Terminal navigate to your working directory with the `dir` command. Then, create a folder with the following command:

```
$ mkdir -p data/db
```

To run the `mongod` service, enter the following command:

```
mongod –dbpath data/db
```

For detailed instructions on running MongoDB suitable to your operating system's environment, please refer to the official documentation at `http://docs.mongodb.org/manual`.

We should get the last two lines of the output similar to the following:

```
Sun Dec  7 00:42:26.063 [websvr] admin web console waiting for
connections on port 28017

Sun Dec  7 00:42:26.064 [initandlisten] waiting for connections on
port 27017
```

This shows that our MongoDB has started successfully and accepts the connections on port 27017. The administrative web interface runs on port 28017 and can be accessed at `http://localhost:28017`.

Getting PyMongo

The PyMongo module contains tools to interact with the MongoDB database using Python. The official page for PyMongo is `https://pypi.python.org/pypi/pymongo/`. This contains the installation instructions that are pretty much similar to installing a regular Python module on Windows, Linux, or Mac OS X. The simplest way to install PyMongo is to download the project source, unzip it to a folder on your local drive, use Terminal to navigate to this folder, and run `python setup.py install` to install it.

Alternatively, if you have pip installed, enter the following command in the Terminal:

```
$ pip install pymongo
```

Running a test connection

Let's do a simple connection check to ensure that our MongoDB service and PyMongo module are installed and running properly with the following Python script:

```
>>> import pymongo
>>> try:
...     client = pymongo.MongoClient("localhost", 27017)
...     print "Connected successfully!!!"
>>> except pymongo.errors.ConnectionFailure, e:
...    print "Could not connect to MongoDB: %s" % e
Connected successfully!!!
```

If the `pymongo.MongoClient` method call is successful, we should get the preceding output. Otherwise, ensure that the MongoDB service is running in the Terminal before executing this script again. Once connected, we can continue with some NoSQL operations.

Getting a database

Within a single data directory at `data/db`, we can create multiple independent databases. With the help of PyMongo, accessing databases can be done with just a single instance of MongoDB.

For example, if we want to have a database called `ticks_db` (which contains an underline character) to store some tick data, we can access this database with PyMongo using the style attributes as follows:

```
>>> ticks_db = client.ticks_db
```

If the database is named in such a way that style attributes are not supported, for example, `ticks-db`, the database can also be accessed as follows:

```
>>> ticks_db = client["ticks-db"]
```

Assigning variables, such as `ticks_db = client.ticks-db` will cause an error!

Getting a collection

A **collection** is a group of documents, stored in a database, that are similar to tables in a relational database. A document accepts various input types described by the **Binary JSON** (**BSON**) specifications, including primitives such as strings and integers, arrays in the form of Python lists, binary strings in the form of UTF-8 encoded-Unicode, `ObjectId`, which is an assigned unique identifier, as well as Python dictionaries as objects. Be aware that each document can contain up to 16 MB of data.

In this example, we are interested in the incoming tick data from the stock ticker AAPL. We can begin to store ticks within a collection named `aapl`:

```
>>> aapl_collection = ticks_db.aapl
```

Similar to accessing databases, should your collection be named with the style attributes that are not supported, say for example, `aapl-collection`, the collection can also be accessed as follows:

```
>>> aapl_collection = ticks_db["aapl-collection"]
```

Note that the collections and databases are created lazily. When objects are inserted into collections and databases that do not yet exist, one will be created.

Inserting a document

Assume that we have a working AAPL market tick data collection system available in Python, and we are interested in storing this data. We can structure the tick data as a Python dictionary and store the ticker name, the time received, the open, high, low, and last prices, as well as the total volume traded information. The following AAPL tick data is just an example of the hypothetical prices received at 10 AM:

```
import datetime as dt
tick = {"ticker": "aapl",
        "time": dt.datetime(2014, 11, 17, 10, 0, 0),
        "open": 115.58,
        "high": 116.08,
        "low": 114.49,
        "last": 114.96,
        "vol": 1900000}
```

We simply use the `insert` function in the collection to insert a dictionary object. A unique key is generated by the `insert` function with a special field named `_id`:

```
>>> tick_id = aapl_collection.insert(tick)
>>> print tick_id
548490486d3ba7178b6c36ba
```

After inserting our first document, the `aapl` collection will be created on the server. To verify this, we can list all of the collections in our database:

```
>>> print ticks_db.collection_names()
[u'system.indexes', u'aapl']
```

Fetching a single document

The most basic query for a document in a collection is the `find_one` function. Using this method, without any parameters, simply gives us the first match, or none if there are no matches. Otherwise, the parameter accepts a dictionary as the filter criteria. The following code contains some `find_one` examples that return the same result. Note that, in the last example, the `ObjectId` attribute needs to be converted from a string in order to access the `_id` field generated by the server:

```
>>> print aapl_collection.find_one()
>>> print aapl_collection.find_one({"time": dt.datetime(2014, 11, 17, \
10, 0, 0)})
>>>
>>> from bson.objectid import ObjectId
>>> print aapl_collection.find_one({"_id": \
>>> ObjectId("548490486d3ba7178b6c36ba")})
{u'last': 114.96, u'vol': 1900000, u'open': 115.58, u'high': 116.08,
u'low': 114.49, u'time': datetime.datetime(2014, 11, 17, 10, 0),
u'_id': ObjectId('548490486d3ba7178b6c36ba'), u'ticker': u'aapl'}
{u'last': 114.96, u'vol': 1900000, u'open': 115.58, u'high': 116.08,
u'low': 114.49, u'time': datetime.datetime(2014, 11, 17, 10, 0),
u'_id': ObjectId('548490486d3ba7178b6c36ba'), u'ticker': u'aapl'}
{u'last': 114.96, u'vol': 1900000, u'open': 115.58, u'high': 116.08,
u'low': 114.49, u'time': datetime.datetime(2014, 11, 17, 10, 0),
u'_id': ObjectId('548490486d3ba7178b6c36ba'), u'ticker': u'aapl'}
```

Deleting documents

The `remove` function removes the documents in the collection that matches the query:

```
>>> aapl_collection.remove()
```

Batch-inserting documents

For batch inserts, the `insert` function accepts a list of comma-separated dictionaries. We will add two more hypothetical tick prices for the next two minutes to our collection:

```
aapl_collection.insert([tick,
                        {"ticker": "aapl",
                         "time": dt.datetime(2014,11,17,10,1,0),
                         "open": 115.58,
                         "high": 116.08,
                         "low": 114.49,
                         "last": 115.00,
                         "vol": 2000000},
                        {"ticker": "aapl",
                         "time": dt.datetime(2014,11,17,10,2,0),
                         "open": 115.58,
                         "high": 116.08,
                         "low": 113.49,
                         "last": 115.00,
                         "vol": 2100000}])
```

Counting documents in the collection

The `count` function can be used on any query to count the number of matches. It can also be used in conjunction with the `find` function:

```
>>> print aapl_collection.count()
>>> print aapl_collection.find({"open": 115.58}).count()
3
3
```

Finding documents

The `find` function is similar to the `find_one` function, except that it returns a list of documents for iteration. Without any parameters, the `find` function simply returns all the items in the collection:

```
>>> for aapl_tick in aapl_collection.find():
...     print aapl_tick
{u'last': 114.96, u'vol': 1900000, u'open': 115.58, u'high': 116.08,
u'low': 114.49, u'time': datetime.datetime(2014, 11, 17, 10, 0),
u'_id': ObjectId('5484943f6d3ba717ca0d26ff'), u'ticker': u'aapl'}
{u'last': 115.0, u'vol': 2000000, u'open': 115.58, u'high': 116.08,
u'low': 114.49, u'time': datetime.datetime(2014, 11, 17, 10, 1),
u'_id': ObjectId('5484943f6d3ba717ca0d2700'), u'ticker': u'aapl'}
{u'last': 115.0, u'vol': 2100000, u'open': 115.58, u'high': 116.08,
u'low': 113.49, u'time': datetime.datetime(2014, 11, 17, 10, 2),
u'_id': ObjectId('5484943f6d3ba717ca0d2701'), u'ticker': u'aapl'}
```

We can also filter our search on the collection of tick data. For example, we are interested in finding the two tick data that arrived before 10:02 AM:

```
>>> cutoff_time = dt.datetime(2014, 11, 17, 10, 2, 0)
>>> for tick in aapl_collection.find(
...         {"time": {"$lt": cutoff_time}}).sort("time"):
...     print tick
{u'last': 114.96, u'vol': 1900000, u'open': 115.58, u'high': 116.08,
u'low': 114.49, u'time': datetime.datetime(2014, 11, 17, 10, 0),
u'_id': ObjectId('5484943f6d3ba717ca0d26ff'), u'ticker': u'aapl'}
{u'last': 115.0, u'vol': 2000000, u'open': 115.58, u'high': 116.08,
u'low': 114.49, u'time': datetime.datetime(2014, 11, 17, 10, 1),
u'_id': ObjectId('5484943f6d3ba717ca0d2700'), u'ticker': u'aapl'}
```

Sorting documents

In the `find.sort` example, we sorted our search results by time in ascending order. We can also do the same in descending order:

```
>>> sorted_ticks = aapl_collection.find().sort(
...     [("time", pymongo.DESCENDING)])
>>> for tick in sorted_ticks:
...     print tick
```

```
{u'last': 115.0, u'vol': 2100000, u'open': 115.58, u'high': 116.08,
u'low': 113.49, u'time': datetime.datetime(2014, 11, 17, 10, 2),
u'_id': ObjectId('548494f16d3ba717d882b83e'), u'ticker': u'aapl'}
{u'last': 115.0, u'vol': 2000000, u'open': 115.58, u'high': 116.08,
u'low': 114.49, u'time': datetime.datetime(2014, 11, 17, 10, 1),
u'_id': ObjectId('548494f16d3ba717d882b83d'), u'ticker': u'aapl'}
{u'last': 114.96, u'vol': 1900000, u'open': 115.58, u'high': 116.08,
u'low': 114.49, u'time': datetime.datetime(2014, 11, 17, 10, 0),
u'_id': ObjectId('548494f16d3ba717d882b83c'), u'ticker': u'aapl'}
```

Conclusion

Using the PyMongo module, we learned how to insert, delete, count, find, and sort tick data on a collection using three ticks' worth of data. Massive write performance, flexibility in the creation of collections, and fast key-value access are some virtues of NoSQL over SQL-based systems. As we continue on our journey in data collection and analysis, we can apply these basic methods to store a continuous stream of tick data for multiple tickers on various collections and databases. Although NoSQL databases may not be the only tick data storage solution available, and may sometimes be passed over for SQL solutions, as we have seen, they integrate nicely with Python and are easy to learn and use. The cross-platform and versatile MongoDB is one of the many NoSQL database products available to store documents in a data-interchangeable format such as the BSON format. Many cloud vendors offer data object storage in a JSON-like structure, similar to those we studied.

Summary

In this chapter, we were introduced to big data and its uses in finance. Big data tools provide the scalability and reliability of analyzing big data in the area of risk and credit analytics, handling data coming in from multiple sources. Apache Hadoop is one popular tool for financial institutions and enterprises in meeting these big data needs.

Apache Hadoop is an open source framework and written in Java. To help us get started quickly with Hadoop, we downloaded a QuickStart VM from Cloudera that comes with CentOS and Hadoop 2.0 running on VirtualBox. The main components in Hadoop are the HDFS file store, YARN, and MapReduce. We learned about Hadoop by writing a map and reduce program in Python to perform a word count on an e-book. Moving on, we downloaded a dataset of the daily prices of a stock and counted the number of percentage intraday price changes. The outputs were taken for further analysis.

Before we begin to manage big data, we will need an avenue to store this data. The nature of digital data is varied, and other means of storing data became the motivation for non-SQL products. One non-relational database language is NoSQL. Because of its simple design, it can also be said to be faster in certain circumstances. One use of NoSQL for finance is the storage of incoming tick data.

We looked at obtaining MongoDB as our NoSQL database server, and the PyMongo module as a way of using Python to interact with MongoDB. After performing a simple test connection to the server with Python, we learned the concepts of databases and collections that are used to store data with PyMongo. Then, a few sample tick data were created and stored as a collection in the BSON format. We investigated how to delete, count, sort, and filter this tick data. These simple operations will enable us to begin storing data continuously and can be later retrieved for further analysis.

In the next chapter, we will take a look at developing an algorithmic trading system with Python.

8
Algorithmic Trading

Algorithmic trading automates the systematic trading process, where orders are executed at the best price possible based on a variety of factors, such as pricing, timing, and volume. Some brokerage firms may offer an **application programming interface** (**API**) as part of their service offering to customers who wish to deploy their own trading algorithms. For developing an algorithmic trading system, it must be highly robust and handle any point of failure during the order execution. Network configuration, hardware, memory management and speed, and user experience are some factors to be considered when designing a system in executing orders. Designing larger systems inevitably add complexity to the framework.

As soon as a position in a market is opened, it is subjected to various types of risk, such as market risk. To preserve the trading capital as much as possible, it is important to incorporate risk management measures to the trading system. Perhaps the most common risk measure used in the financial industry is the **value-at-risk** (**VaR**) technique. We will discuss the beauty and flaws of VaR, and how it can be incorporated into our trading system that we will develop in this chapter.

In this chapter, we will cover the following topics:

- An overview of algorithmic trading
- List of brokers and system vendors with public API
- Choosing a programming language for a trading system
- Setting up API access on Interactive Brokers (IB) trading platform
- Using the IbPy module to interact with IB Trader WorkStation (TWS)
- Implementing a mean-reverting algorithmic trading strategy
- Setting up API access on OANDA with fxTrade Practice platform
- Using the oandapy module to interact with OANDA's REST API

- Implementing a trend-following algorithmic trading strategy
- Introduction to VaR for risk management in our trading system
- Performing VaR calculation in Python with data from Yahoo! Finance

Introduction to algorithmic trading

In the 1990s, exchanges had already begun to use electronic trading systems. By 1997, 44 exchanges worldwide used automated systems for trading futures and options with more exchanges in the process of developing automated technology. Exchanges such as the **Chicago Board of Trade** (**CBOT**) and the **London International Financial Futures and Options Exchange** (**LIFFE**) used their electronic trading systems as an after-hours complement to traditional open outcry trading in pits, giving traders 24-hour access to the exchange's risk management tools. With improvements in technology, technology-based trading became less expensive, fueling the growth of trading platforms that are faster and powerful. Higher reliability of order execution and lower rates of message transmission error deepened the reliance of technology by financial institutions. The majority of asset managers, proprietary traders, and market makers have since moved from the trading pits to electronic trading floors.

As systematic or computerized trading became more commonplace, speed became the most important factor in determining the outcome of a trade. Quants utilizing sophisticated fundamental models are able to recompute fair values of trading products on the fly and execute trading decisions, enabling them to reap profits at the expense of fundamental traders using traditional tools. This gave way to the term **high-frequency trading** (**HFT**) that relies on fast computers to execute the trading decisions before anyone else can. HFT has evolved into a billion-dollar industry.

Algorithmic trading refers to the automation of the systematic trading process, where the order execution is heavily optimized to give the best price possible. It is not part of the portfolio allocation process.

Banks, hedge funds, brokerage firms, clearing firms, and trading firms typically have their servers placed right next to the electronic exchange to receive the latest market prices and to perform the fastest order execution where possible. They bring enormous trading volumes to the exchange. Anyone who wishes to participate in low-latency, high-volume trading activities, such as complex event processing or capturing fleeting price discrepancies, by acquiring exchange connectivity may do so in the form of co-location, where his or her server hardware can be placed on a rack right next to the exchange for a fee.

The **Financial Information Exchange** (**FIX**) protocol is the industry standard for electronic communications with the exchange from the private server for **direct market access** (**DMA**) to real-time information. C++ is the common choice of programming language for trading over the FIX protocol, though other languages, such as .NET framework common language and Java can be used. Before creating an algorithmic trading platform, you would need to assess various factors, such as speed and ease of learning before deciding on a specific language for the purpose.

Brokerage firms would provide a trading platform of some sort to their customers for them to execute orders on selected exchanges in return for the commission fees. Some brokerage firms may offer an API as part of their service offering to technically inclined customers who wish to run their own trading algorithms. In most circumstances, customers may also choose from a number of commercial trading platforms offered by third-party vendors. Some of these trading platforms may also offer API access to route orders electronically to the exchange. It is important to read the API documentation beforehand to understand the technical capabilities offered by your broker and to formulate an approach in developing an algorithmic trading system.

List of trading platforms with public API

The following table lists some brokers and trading platform vendors who have their API documentation publicly available:

Broker/ vendor	**URL**	**Programming languages supported**
Interactive Brokers	`https://www.interactivebrokers.com/en/index.php?f=1325`	C++, Posix C++, Java, and Visual Basic for ActiveX
E*Trade	`https://developer.etrade.com`	Java, PHP, and C++
IG	`http://labs.ig.com/`	REST, Java, FIX, and Microsoft .NET Framework 4.0
Tradier	`https://developer.tradier.com`	Java, Perl, Python, and Ruby
TradeKing	`https://developers.tradeking.com`	Java, Node.js, PHP, R, and Ruby
Cunningham trading systems	`http://www.ctsfutures.com/wiki/T4%20API%2040.MainPage.ashx`	Microsoft .NET Framework 4.0

Broker/ vendor	URL	Programming languages supported
CQG	`http://cqg.com/Products/CQG-API.aspx`	C#, C++, Excel, MATLAB, and VB.NET
Trading technologies	`https://developer.tradingtechnologies.com`	Microsoft .NET Framework 4.0
OANDA	`http://developer.oanda.com`	REST, Java, FIX, and MT4

Which is the best programming language to use?

With many choices of programming languages available to interface with brokers or vendors, the question that comes naturally to anyone starting out in algorithmic trading platform development is: which language should I use?

Well, the short answer is that there is really no best programming language. How your product will be developed, the performance metrics to follow, the costs involved, latency threshold, risk measures, and the expected user interface are pieces of the puzzle to be taken into consideration. The risk manager, execution engine, and portfolio optimizer are some major components that will affect the design of your system. Your existing trading infrastructure, choice of operating system, programming language compiler capability, and available software tools poses further constraints on the system design, development, and deployment.

System functionalities

It is important to define the outcomes of your trading system. An outcome could be a research-based system that might be more concerned with obtaining high-quality data from data vendors, performing computations or running models, and evaluating a strategy through signal generation. Part of the research component might include a data-cleaning module or a backtesting interface to run a strategy with theoretical parameters over historical data. The CPU speed, memory size, and bandwidth are factors to be considered while designing our system.

Another outcome could be an execution-based system that is more concerned with risk management and order handling features to ensure timely execution of multiple orders. The system must be highly robust and handle any point of failure during the order execution. As such, network configuration, hardware, memory management and speed, and user experience are some factors to be considered when designing a system in executing orders.

A system may contain one or more of these functionalities. Designing larger systems inevitably add complexity to the framework. It is recommended that you choose one or more programming languages that can address and balance the development speed, ease of development, scalability, and reliability of your trading system.

Algorithmic trading with Interactive Brokers and IbPy

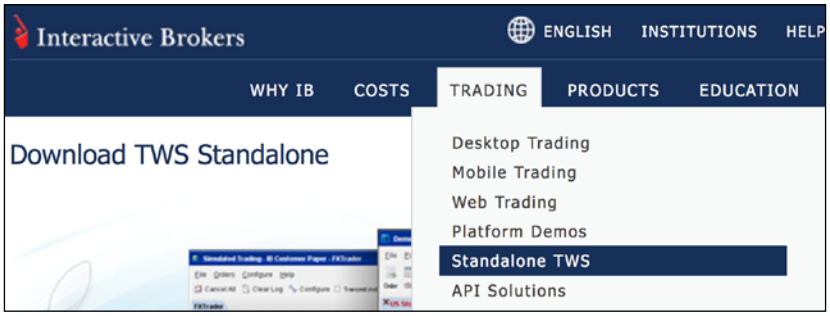
In this section, we will build a working algorithmic trading platform that will authenticate with **Interactive Brokers** (**IB**) and log in, retrieve the market data, and send orders. IB is one of the most popular brokers in the trading community and has a long history of API development. There are plenty of articles on the use of the API available on the Web. IB serves clients ranging from hedge funds to retail traders. Although the API does not support Python directly, Python wrappers such as IbPy are available to make the API calls to the IB interface. The IB API is unique to its own implementation, and every broker has its own API handling methods. Nevertheless, the documents and sample applications provided by your broker would demonstrate the core functionality of every API interface that can be easily integrated into an algorithmic trading system if designed properly.

Getting Interactive Brokers' Trader WorkStation

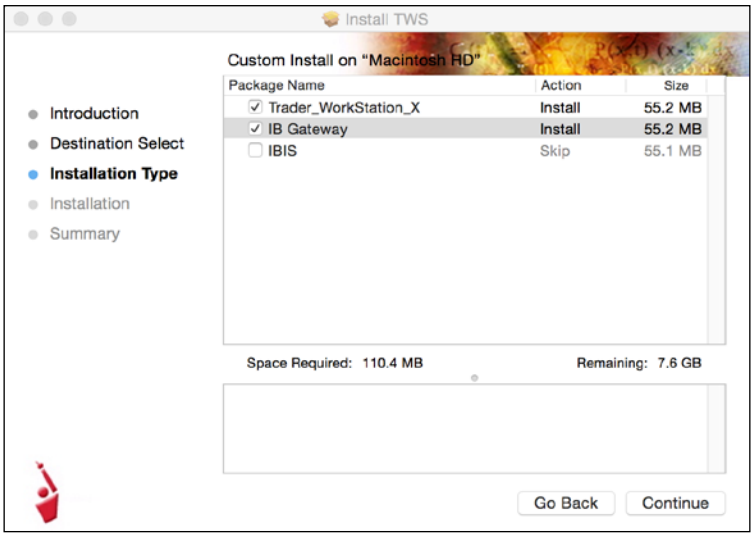
The official page for IB is `http://www.interactivebrokers.com`. Here, you can find a wealth of information regarding trading and investing for retail and institutional traders. In this section, we will take a look at how to get the **Trader WorkStation X** (**TWS**) installed and running on your local workstation before setting up an algorithmic trading system using Python. Note that we will perform simulated trading on a demonstration account. If your trading strategy turns out to be profitable, head to the **OPEN AN ACCOUNT** section of the IB website to open a live trading account. Rules, regulations, market data fees, exchange fees, commissions, and other conditions are subjected to the broker of your choice. In addition, market conditions are vastly different from the simulated environment. You are encouraged to perform extensive testing on your algorithmic trading system before running on live markets.

The following key steps describe how to install TWS on your local workstation, log in to the demonstration account, and set it up for API use:

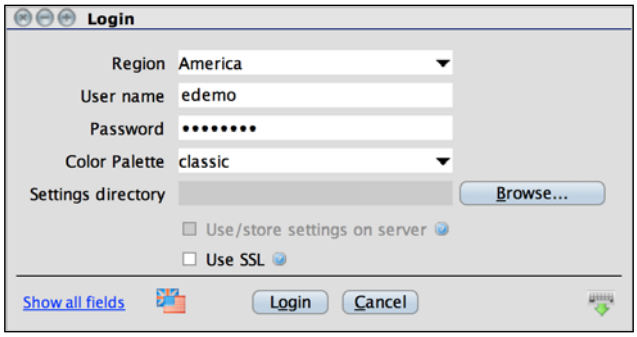
1. From IB's official website, navigate to **TRADING**, then select **Standalone TWS**. Choose the installation executable that is suitable for your local workstation. TWS runs on Java; therefore, ensure that Java runtime plugin is already installed on your local workstation. Refer to the following screenshot:



2. When prompted during the installation process, choose **Trader_WorkStation_X** and **IB Gateway** options. The Trader WorkStation X (TWS) is the trading platform with full order management functionality. The IB Gateway program accepts and processes the API connections without any order management features of the TWS. We will not cover the use of the IB Gateway, but you may find it useful later. Select the destination directory on your local workstation where TWS will place all the required files, as shown in the following screenshot:

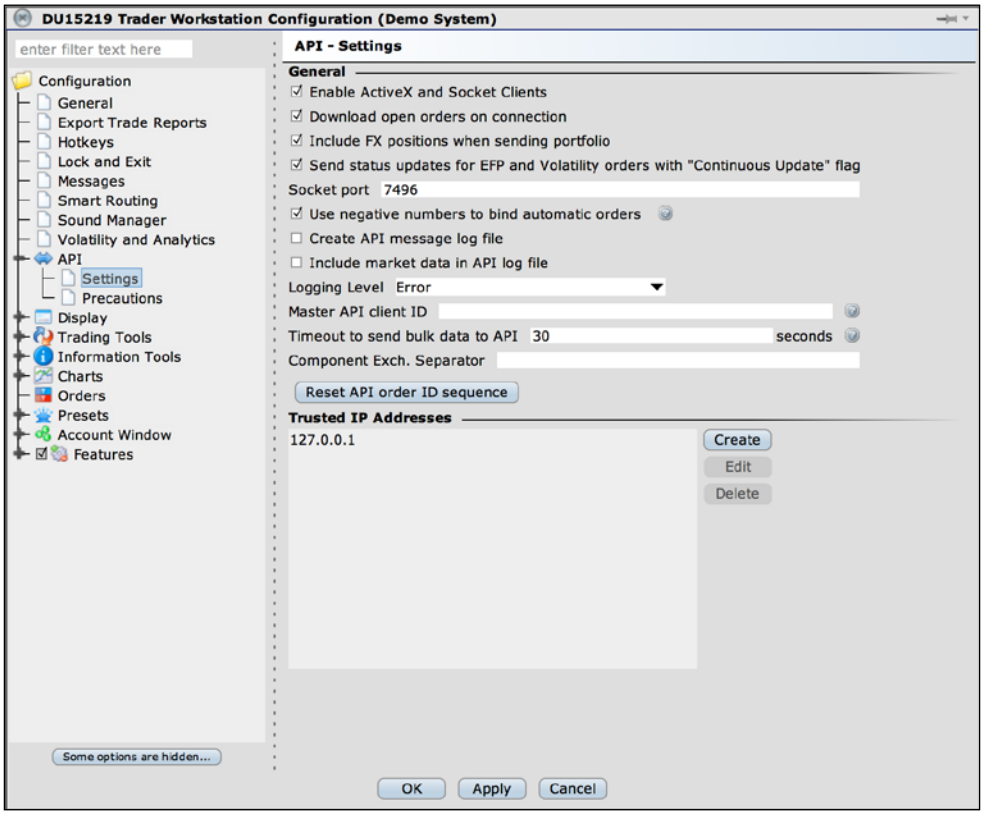


3. When the installation is completed, a TWS shortcut icon will appear together with your list of installed applications. Double-click on the icon to start the TWS program.
4. When TWS starts, you will be prompted to enter your login credentials. To log in to the demonstration account, type `edemo` in the username field and `demouser` in the password field, as shown in the following screenshot:



5. Once we have managed to load our demo account on TWS, we can now set up its API functionality. On the toolbar, click on **Configure**:

6. Under the **Configuration** tree, open the API node to reveal further options. Select **Settings**. Note that **Socket port** is **7496**, and we added the IP address of our workstation housing our algorithmic trading system to the list of trusted IP addresses, which in this case is **127.0.0.1**. Ensure that the **Enable ActiveX and Socket Clients** option is selected to allow the socket connections to TWS:



7. Click on **OK** to save all the changes. TWS is now ready to accept orders and market data requests from our algorithmic trading system.

Getting IbPy – the IB API wrapper

IbPy is an add-on module for Python that wraps the IB API. It is open source and can be found at `https://github.com/blampe/IbPy`. Head to this URL and download the source files. Unzip the source folder, and use Terminal to navigate to this directory. Type `python setup.py install` to install IbPy as part of the Python runtime environment.

The use of IbPy is similar to the API calls, as documented on the IB website. The documentation for IbPy is at `https://code.google.com/p/ibpy/w/list`.

A simple order routing mechanism

In this section, we will start interacting with TWS using Python by establishing a connection and sending out a market order to the exchange.

Once IbPy is installed, import the following necessary modules into our Python script:

```
from ib.ext.Contract import Contract
from ib.ext.Order import Order
from ib.opt import Connection
```

Next, implement the logging functions to handle calls from the server. The `error_handler` method is invoked whenever the API encounters an error, which is accompanied with a message. The `server_handler` method is dedicated to handle all the other forms of returned API messages. The `msg` variable is a type of an `ib.opt.message` object and references the method calls, as defined by the IB API `EWrapper` methods. The API documentation can be accessed at `https://www.interactivebrokers.com/en/software/api/api.htm`. The following is the Python code for the `server_handler` method:

```
def error_handler(msg):
    print "Server Error:", msg
def server_handler(msg):
    print "Server Msg:", msg.typeName, "-", msg
```

We will place a sample order of the stock AAPL. The contract specifications of the order are defined by the `Contract` class object found in the `ib.ext.Contract` module. We will create a method called `create_contract` that returns a new instance of this object:

```
def create_contract(symbol, sec_type, exch, prim_exch, curr):
    contract = Contract()
    contract.m_symbol = symbol
    contract.m_secType = sec_type
    contract.m_exchange = exch
    contract.m_primaryExch = prim_exch
    contract.m_currency = curr
    return contract
```

The `Order` class object is used to place an order with TWS. Let's define a method called `create_order` that will return a new instance of the object:

```
def create_order(order_type, quantity, action):
    order = Order()
    order.m_orderType = order_type
    order.m_totalQuantity = quantity
    order.m_action = action
    return order
```

After the required methods are created, we can then begin to script the main functionality. Let's initialize the required variables:

```
if __name__ == "__main__":
    client_id = 100
    order_id = 1
    port = 7496
    tws_conn = None
```

Note that the `client_id` variable is our assigned integer that identifies the instance of the client communicating with TWS. The `order_id` variable is our assigned integer that identifies the order queue number sent to TWS. Each new order requires this value to be incremented sequentially. The port number has the same value as defined in our API settings of TWS earlier. The `tws_conn` variable holds the connection value to TWS. Let's initialize this variable with an empty value for now.

Let's use a `try` block that encapsulates the `Connection.create` method to handle the socket connections to TWS in a graceful manner:

```
try:
    # Establish connection to TWS.
    tws_conn = Connection.create(port=port,
                                 clientId=client_id)
    tws_conn.connect()

    # Assign error handling function.
    tws_conn.register(error_handler, 'Error')

    # Assign server messages handling function.
    tws_conn.registerAll(server_handler)

finally:
    # Disconnect from TWS
    if tws_conn is not None:
        tws_conn.disconnect()
```

The `port` and `clientId` parameter fields define this connection. After the connection instance is created, the `connect` method will try to connect to TWS.

When the connection to TWS has successfully opened, it is time to register listeners to receive notifications from the server. The `register` method associates a function handler to a particular event. The `registerAll` method associates a handler to all the messages generated. This is where the `error_handler` and `server_handler` methods declared earlier will be used for this occasion.

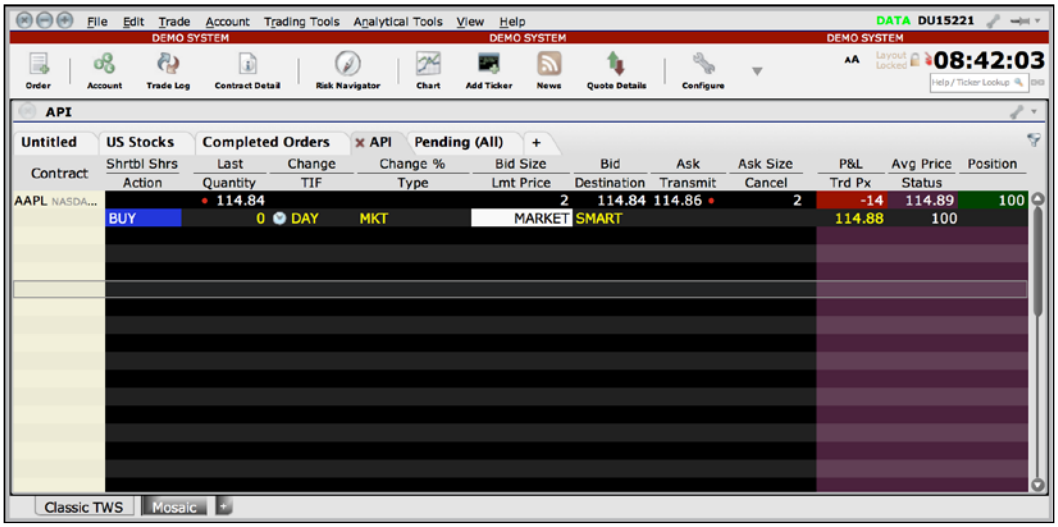
Before sending our very first order of 100 shares of AAPL to the exchange, we will call the `create_contract` method to create a new contract object for AAPL. Then, we will call the `create_order` method to create a new `Order` object, to go long 100 shares. Finally, we will call the `placeOrder` method of the `Connection` class to send out this order to TWS:

```
# Create a contract for AAPL stock using SMART order routing.
aapl_contract = create_contract('AAPL',
                                'STK',
                                'SMART',
                                'SMART',
                                'USD')

# Go long 100 shares of AAPL
aapl_order = create_order('MKT', 100, 'BUY')

# Place order on IB TWS.
tws_conn.placeOrder(order_id, aapl_contract, aapl_order)
```

That's it! Let's run our Python script. We should get a similar output as follows:

```
Server Error: <error id=-1, errorCode=2104, errorMsg=Market data farm
connection is OK:ibdemo>
Server Response: error, <error id=-1, errorCode=2104, errorMsg=Market
data farm connection is OK:ibdemo>
Server Version: 75
TWS Time at connection:20141210 23:14:17 CST
Server Msg: managedAccounts - <managedAccounts accountsList=DU15200>
Server Msg: nextValidId - <nextValidId orderId=1>
Server Error: <error id=-1, errorCode=2104, errorMsg=Market data farm
connection is OK:ibdemo>
Server Msg: error - <error id=-1, errorCode=2104, errorMsg=Market data
farm connection is OK:ibdemo>
Server Error: <error id=-1, errorCode=2107, errorMsg=HMDS data farm
connection is inactive but should be available upon demand.demohmds>
Server Msg: error - <error id=-1, errorCode=2107, errorMsg=HMDS data farm
connection is inactive but should be available upon demand.demohmds>
```

Basically, what the error messages say is that there are no errors and the connections are OK. Should the simulated order be executed successfully during market trading hours, the trade will be reflected in TWS:



The full source code of our implementation is given as follows:

```
""" A Simple Order Routing Mechanism """
from ib.ext.Contract import Contract
from ib.ext.Order import Order
from ib.opt import Connection

def error_handler(msg):
    print "Server Error:", msg

def server_handler(msg):
    print "Server Msg:", msg.typeName, "-", msg

def create_contract(symbol, sec_type, exch, prim_exch, curr):
    contract = Contract()
    contract.m_symbol = symbol
    contract.m_secType = sec_type
    contract.m_exchange = exch
    contract.m_primaryExch = prim_exch
    contract.m_currency = curr
    return contract

def create_order(order_type, quantity, action):
```

```
    order = Order()
    order.m_orderType = order_type
    order.m_totalQuantity = quantity
    order.m_action = action
    return order

if __name__ == "__main__":
    client_id = 1
    order_id = 119
    port = 7496
    tws_conn = None
    try:
        # Establish connection to TWS.
        tws_conn = Connection.create(port=port,
                                     clientId=client_id)
        tws_conn.connect()

        # Assign error handling function.
        tws_conn.register(error_handler, 'Error')

        # Assign server messages handling function.
        tws_conn.registerAll(server_handler)

        # Create AAPL contract and send order
        aapl_contract = create_contract('AAPL',
                                        'STK',
                                        'SMART',
                                        'SMART',
                                        'USD')

        # Go long 100 shares of AAPL
        aapl_order = create_order('MKT', 100, 'BUY')

        # Place order on IB TWS.
        tws_conn.placeOrder(order_id, aapl_contract, aapl_order)

    finally:
        # Disconnect from TWS
        if tws_conn is not None:
            tws_conn.disconnect()
```

Building a mean-reverting algorithmic trading system

In the previous section, we established a connection to IB TWS and sent a market order of 100 shares. In this section, we will add logic functions to buy or sell a number of shares, read tick data, and track our positions. In essence, we will try to create a simple, fully automated algorithmic trading system.

Setting up the main program

Since our code might get a little complicated, let's tidy up and put everything into a class named `AlgoSystem`. We will import the following modules:

```
from ib.ext.Contract import Contract
from ib.ext.Order import Order
from ib.opt import Connection, message
import time
import pandas as pd
import datetime as dt
```

In the initialization section of our class, declare the following variables:

```
def __init__(self, symbol, qty, resample_interval,
             averaging_period=5, port=7496):
    self.client_id = 1
    self.order_id = 1
    self.qty = qty
    self.symbol_id, self.symbol = 0, symbol
    self.resample_interval = resample_interval
    self.averaging_period = averaging_period
    self.port = port
    self.tws_conn = None
    self.bid_price, self.ask_price = 0, 0
    self.last_prices = pd.DataFrame(columns=[self.symbol_id])
    self.average_price = 0
    self.is_position_opened = False
    self.account_code = None
    self.unrealized_pnl, self.realized_pnl = 0, 0
    self.position = 0
```

The first argument of the constructor accepts the ticker symbol that we will trade. The integer identifier for the ticker is set as `0` in the `symbol_id` variable. The `qty` variable contains the quantity of shares to trade. The `resample_interval` variable defines the resampling period of the time series data, which we will use later, while the `averaging_period` variable defines a number of periods of the resampled series to be taken into account for calculating the average of prices. The `last_prices` variable is a pandas `DataFrame` object used to store all of our last prices pertaining to the stock ticker.

For our main program, the `start` method is the entry point that lasts during the lifetime of the running trading system:

```
def start(self):
    try:
        self.connect_to_tws()
        self.register_callback_functions()
        self.request_market_data(self.symbol_id, self.symbol)
        self.request_account_updates(self.account_code)

        while True:
            time.sleep(1)

    except Exception, e:
        print "Error:", e
        self.cancel_market_data(self.symbol)

    finally:
        print "disconnected"
        self.disconnect_from_tws()
```

The `connect_to_tws` method establishes the socket connection to TWS, as discussed in the previous section. The `register_callback_functions` method assigns functions that will handle the server responses, which we will discuss later. The `request_market_data` method will notify the server to begin streaming data to our trading system. Similarly, the `request_account_updates` method will notify the server to begin streaming the position information to our trading system. The `account_code` variable contains the account number assigned by our broker and would be assigned upon connection to TWS.

After all the necessary function calls to TWS have been made, an infinite *while* loop keeps our trading system program running in the background while responding to market events. Should our script be stopped, the `cancel_market_data` method gracefully terminates the market data streaming functionality before closing the connection to TWS.

Let's take a look at the implementation of the `register_callback_functions` method:

```
def register_callback_functions(self):
    # Assign server messages handling function.
    self.tws_conn.registerAll(self.server_handler)

    # Assign error handling function.
    self.tws_conn.register(self.error_handler, 'Error')

    # Register market data events.
    self.tws_conn.register(self.tick_event,
                           message.tickPrice,
                           message.tickSize)
```

Note that an additional `register` method has been added to listen for the `tickPrice` and `tickSize` objects of the server message object. The `tick_event` method will handle the incoming tick data.

The `request_market_data` method requests TWS to start streaming the market data for a particular stock ticker symbol. The `reqMktData` method accepts an integer identifier of the symbol and a `Contract` object. The generic tick type parameter is empty for this occasion, and the snapshot market data is not required, as indicated by a `False` value. The use of this method corresponds to the `reqMktData` method documented on the IB API page.

The `time.sleep` method is added to allow sufficient communication delay and ensure completion of the connection to TWS:

```
def request_market_data(self, symbol_id, symbol):
    contract = self.create_contract(symbol,
                                    'STK',
                                    'SMART',
                                    'SMART',
                                    'USD')
    self.tws_conn.reqMktData(symbol_id, contract, '', False)
    time.sleep(1)
```

Requesting for account updates is also performed in the same fashion. If the first parameter is set to `True`, the client will start receiving account and portfolio updates. Otherwise, the client will stop receiving this information:

```
def request_account_updates(self, account_code):
    self.tws_conn.reqAccountUpdates(True, account_code)
```

Handling events

Let's modify the `error_handler` method to print the error messages only:

```
def error_handler(self, msg):
    if msg.typeName == "error" and msg.id != -1:
        print "Server Error:", msg
```

The `server_handler` method handles all the other events, including position and account updates, except market data. At this moment, we are only interested in the position, unrealized profit and loss, and realized profit and loss, with regards to our current stock ticker. This information can be obtained from the tick data of type `updatePortfolio`. Should other positions from different stock tickers be available, this information can also be obtained here. The `nextValidId` and the `managedAccounts` tick data types contain information on the next required identification number and the account code respectively. We will ignore the rest of the messages at this moment:

```
def server_handler(self, msg):
    if msg.typeName == "nextValidId":
        self.order_id = msg.orderId
    elif msg.typeName == "managedAccounts":
        self.account_code = msg.accountsList
    elif msg.typeName == "updatePortfolio" \
            and msg.contract.m_symbol == self.symbol:
        self.unrealized_pnl = msg.unrealizedPNL
        self.realized_pnl = msg.realizedPNL
        self.position = msg.position
    elif msg.typeName == "error" and msg.id != -1:
        return
```

The `tick_event` method handles the market data that is subscribed to TWS, as defined by the `tickPrice` and `tickSize` types of tick data. In the `msg` variable of type `message` object, a field value of `1` indicates that the price received is the market bid price, a value of `2` indicates that the price received is the market ask price, and a value of `3` indicates the last traded price. In this system, we are only interested in using the last price. Further use of the last price is described in the next section.

Also, on every market tick information received, we will perform the trading logic in the `perform_trade_logic` method:

```
def tick_event(self, msg):
    if msg.field == 1:
        self.bid_price = msg.price
    elif msg.field == 2:
        self.ask_price = msg.price
```

```
        elif msg.field == 4:
            self.last_prices.loc[dt.datetime.now()] = msg.price
            resampled_prices = \
                self.last_prices.resample(self.resample_interval,
                                          how='last',
                                          fill_method="ffill")
        self.average_price = resampled_prices.tail(
            self.averaging_period).mean()[0]
        self.perform_trade_logic()
```

Implementing the mean-reverting algorithm

Let's implement a simple mean-reverting strategy to our algorithmic trading system. The mean-reversion algorithm is one of the most used methods in trading studies.

Suppose we believe that in normal market conditions, the average price of a stock, given a certain number of prior periods will lie between the bid price and the offer price. When the market prices deviate from the average price, perhaps due to some heavy market forces, we believe that the market would revert back to the long-term mean price level. When this happens, we would like to open a position using a market order. We can also close our position in the same manner.

In the `tick_event` method, the received prices are used in the trading logic. Last price values are stored in a pandas `DataFrame` object indexed by the time the tick data is received. The time difference between two consecutive ticks could be as little as a fraction of a second in volatile market conditions. We would be doing some sort of high-frequency data management but not true high-frequency trading. The first step in studying high-frequency data is to standardize the timestamps by resampling. The pandas module contains a `resample` method to sample the time series prices at regular intervals:

```
self.last_prices.resample(self.resample_interval,
                          how='last',
                          fill_method="ffill")
```

This is the code to be used for resampling the time series of last prices. The first parameter defines the resampling interval, currently stated as thirty-second intervals. The `how` parameter lets us choose how the resampling will take place. Here, we chose to take the last price at the end of every interval using the `last` value. The `fill_method` parameter lets us choose how to populate the empty values encountered in the resampled pandas time series. Specify this parameter with the `ffill` value will forward fill-empty resampled values with the preceding values, where applicable.

The `average_price` variable stores the mean of the last traded prices for the past five-minute intervals, as defined by `averaging_period`. With 30 seconds per interval, we are looking at the average resampled price for the past two and a half minutes:

```
self.average_price = resampled_prices.tail(
    self.averaging_period).mean()[0]
```

The implementation of the `perform_trade_logic` function is given as follows:

```
def perform_trade_logic(self):
    # Is buying at the market lower than the average price?
    is_buy_signal = self.ask_price < self.average_price

    # Is selling at the market higher than the average price?
    is_sell_signal = self.bid_price > self.average_price

    # Print signal values on every tick
    print dt.datetime.now(), \
        " BUY/SELL? ", is_buy_signal, "/", is_sell_signal, \
        " Avg:", self.average_price

    # Use generated signals, if any, to open a position
    if self.average_price != 0 \
        and self.bid_price != 0 \
        and self.ask_price != 0 \
        and self.position == 0 \
            and not self.is_position_opened:

        if is_sell_signal:
            self.place_market_order(self.symbol, self.qty , False)
            self.is_position_opened = True

        elif is_buy_signal:
            self.place_market_order(self.symbol, self.qty , True)
            self.is_position_opened = True

    # If position is already opened, use generated signals
    # to close the position.
    elif self.is_position_opened:

        if self.position > 0 and is_sell_signal:
            self.place_market_order(self.symbol, self.qty , False)
            self.is_position_opened = False
```

```
            elif self.position <0 and is_buy_signal:
                self.place_market_order(self.symbol, self.qty , True)
                self.is_position_opened = False

            # When the position is open, keep track of our
            # unrealized and realized P&Ls
            self.monitor_position()
```

During the entire period, when our trading system is active, we would also want to print some information to the console, including signal variables and trade performance. The `monitor_position` method helps you achieve this purpose.

Tracking our positions

Our mean-reverting trading system is now complete. Let's run the program from the main function by creating an instance of `AlgoSystem` to trade 100 shares of the stock FB:

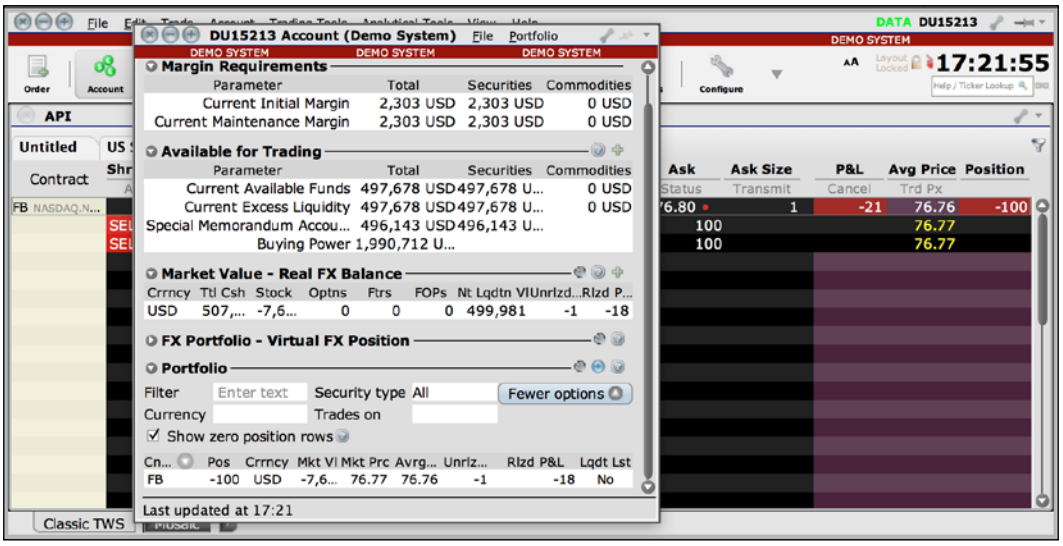
```
>>> if __name__ == "__main__":
>>>     system = AlgoSystem("FB", 100, "30s", 5)
>>>     system.start()
```

We should get an output that looks something like this:

```
Server Version: 75
TWS Time at connection:20141212 17:20:11 CST
2014-12-12 17:20:25.394539  BUY/SELL?  False / False  Avg: 76.77
2014-12-12 17:20:25.394655  BUY/SELL?  False / False  Avg: 76.77
2014-12-12 17:20:25.647989  BUY/SELL?  False / False  Avg: 76.77
2014-12-12 17:20:25.650204  BUY/SELL?  True / False  Avg: 76.81
Placed order 100 of FB to BUY
2014-12-12 17:20:25.652497  BUY/SELL?  True / False  Avg: 76.81
…
2014-12-12 17:20:28.400413  BUY/SELL?  False / False  Avg: 76.77
Position:100 UPnL:-4.0 RPnL:-4.0
2014-12-12 17:20:28.722000  BUY/SELL?  False / False  Avg: 76.77
Position:100 UPnL:-4.0 RPnL:-4.0
2014-12-12 17:20:28.724448  BUY/SELL?  True / False  Avg: 76.8
…
2014-12-12 17:20:56.471880  BUY/SELL?  False / False  Avg: 76.775
Position:100 UPnL:-4.0 RPnL:-4.0
```

```
2014-12-12 17:20:56.684852  BUY/SELL?  False / False  Avg: 76.775
Position:100 UPnL:-4.0 RPnL:-4.0
2014-12-12 17:20:56.687029  BUY/SELL?  False / True  Avg: 76.74
Placed order 100 of FB to SELL
2014-12-12 17:20:56.688781  BUY/SELL?  False / True  Avg: 76.74
2014-12-12 17:20:57.519985  BUY/SELL?  False / True  Avg: 76.74
```

In TWS, we can track our positions from the **Account** section:



It looks like more work needs to be done to make our algorithm profitable. Most trading systems implement a combination of different signal indicators. Hedging is also performed so that our positions will not be exposed to too much risk. In other words, a robust trading system requires detailed planning and extensive backtesting in reducing the probability of losses and increasing the probability of profits. Many commercial trading platforms come with a host of built-in advanced features. It is ultimately up to you as a trader to choose the trading platform that lets you best implement your trading strategy and to suit your individual style of trading.

The full source code for the `AlgoSystem` class is available at the Packt Publishing website.

Forex trading with OANDA API

In the previous sections, we implemented a trading system by interfacing with Interactive Brokers' Trader WorkStation X through the socket connections over a single port. However, many other brokers offer different choices of hooking up customized trading software over an API. In this section, we will learn how to interface our trading strategy with OANDA's REST API. OANDA is a major player in the foreign exchange (forex) business servicing retail investors. We will use a trend-following strategy to trade forex products.

What is REST?

REST stands for **Representational State Transfer**. It refers to web service APIs for transferring data over HTTP using the `GET`, `PUT`, `POST`, or `DELETE` methods.

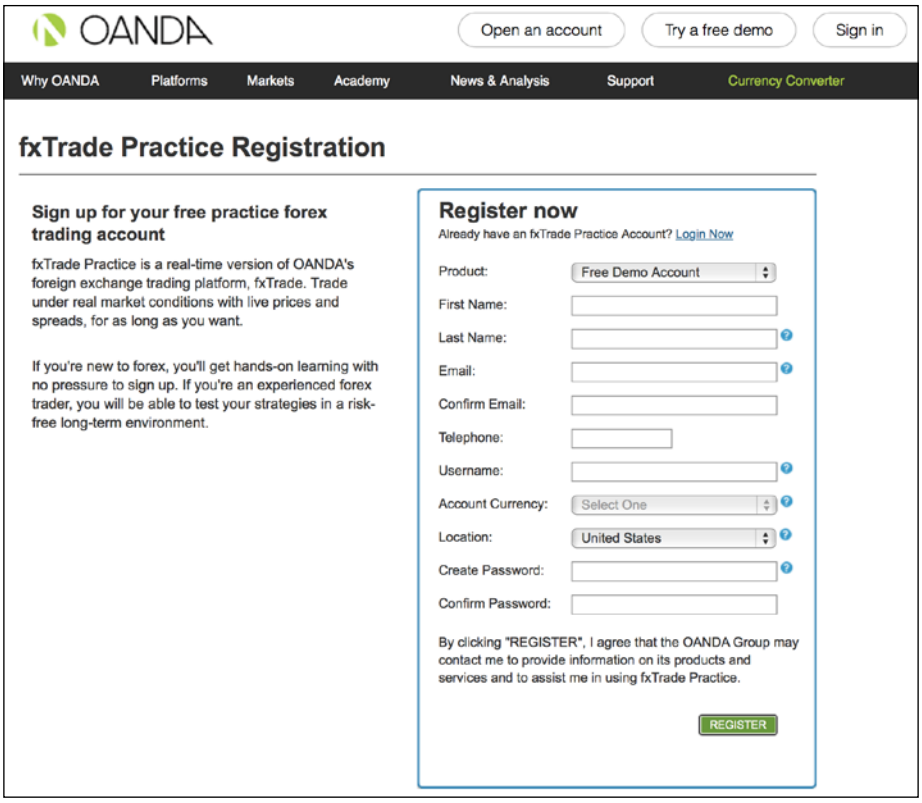
With the REST API, we can stream the market data and trade the markets using any programming language that supports data transfer over HTTP connections along with a **JavaScript Object Notation** (**JSON**) parser.

Note that REST connections are stateless—the sender or receiver does not store information for future use. The sender does not expect an acknowledgement of data received; the receiver accepts the data packets without any prior connection setup.

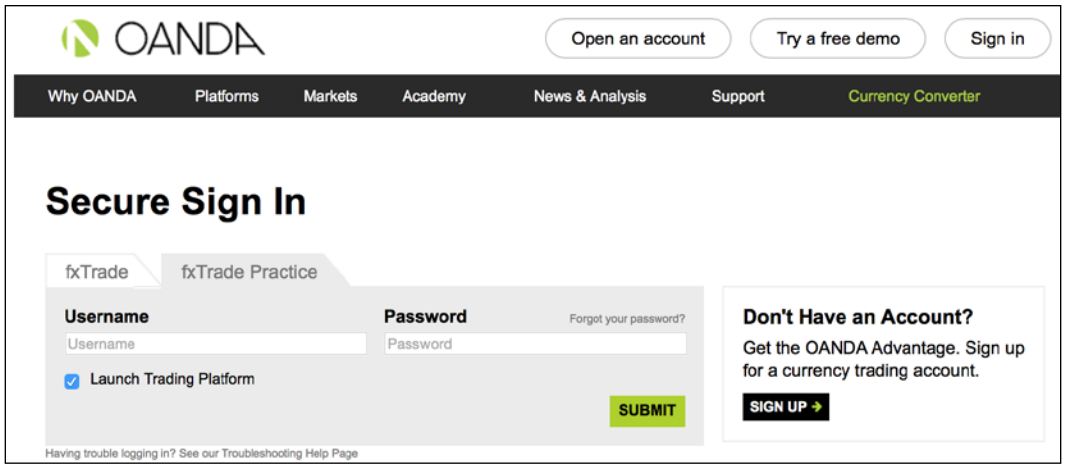
Setting up an OANDA account

OANDA offers a free practice account for testing your trading system. Once you are ready for the transit to trading on a live account, all that your trading system needs in order to be configured is the associated account identifier and the access token key.

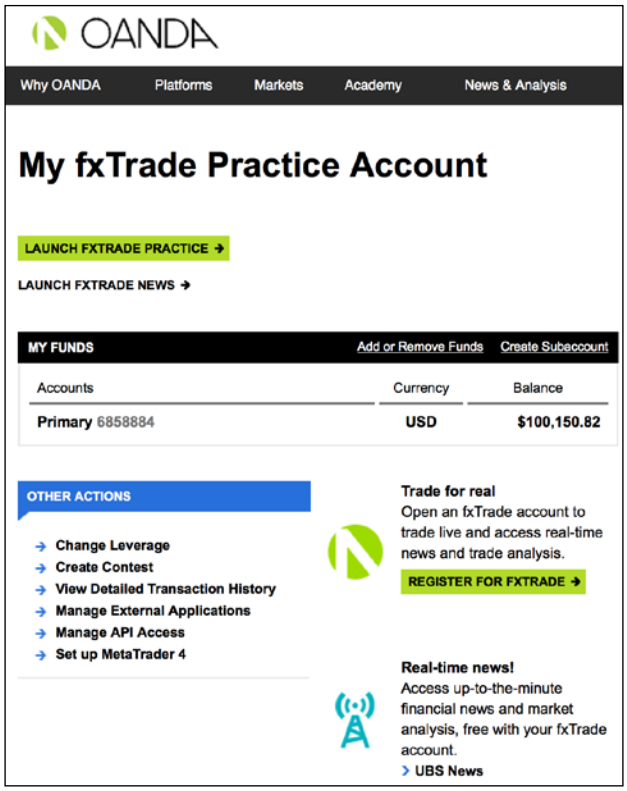
To create your practice account, simply navigate to `http://www.oanda.com` in your web browser and select **Try a free demo**. Alternatively, the direct sign-up link is `https://fxtrade.oanda.com/your_account/fxtrade/register/game/signup`. You should see a sign-up page similar to the following screenshot:



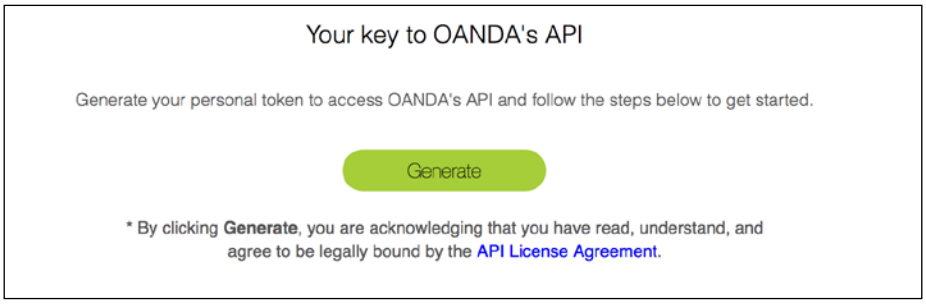
Once you have activated your account, you may click on **Sign in** to visit the login page. Before entering your login credentials, be sure to select the **fxTrade Practice** tab, as shown in the following screenshot:



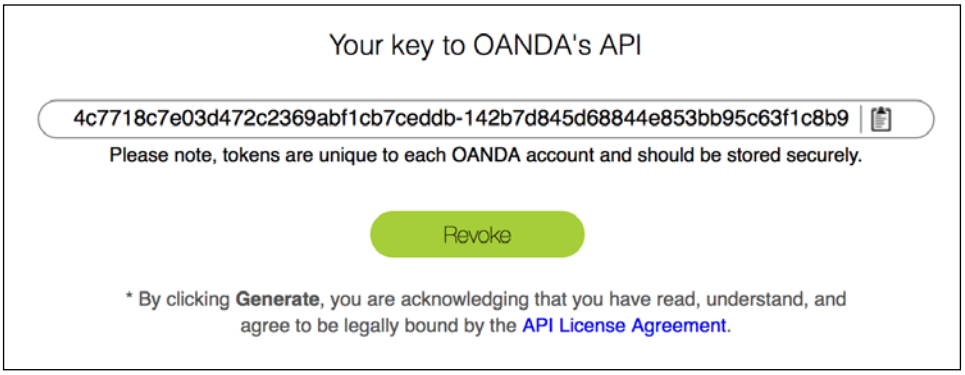
On successful sign in, you will be able to view your practice account management page, as shown in the following screenshot. Under the **MY FUNDS** section, take a note of the 7-digit number next to the primary account field. This will be your account identifier. The account identifier shown in this example is **6858884**:



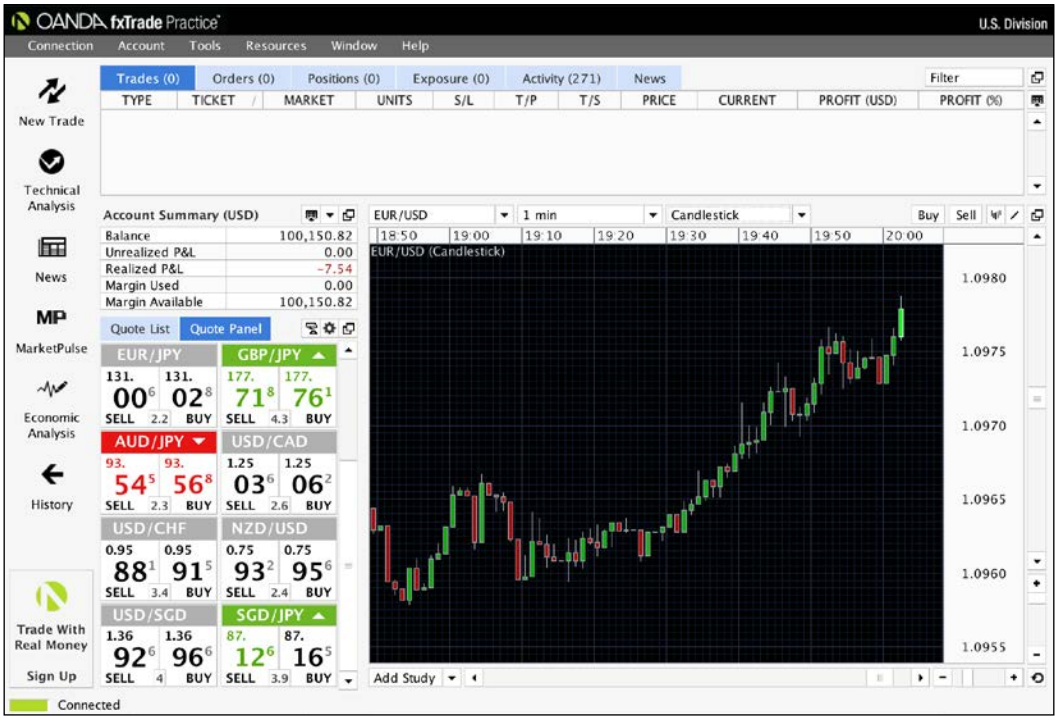
To obtain an API access token for your applications, under **OTHER ACTIONS** section, at the bottom-left of the page, select **Manage API Access**. You will be brought to a page similar to this:



At this stage, click on **Generate** to obtain an API access token. An API access token is unique and can only be generated once. Make sure to note down this token for later use:



Return to the practice account management page. Here, you can also launch **fxTrade Practice**. fxTrade is one of the trading platforms offered by OANDA. It can be used on any operating system, such as Windows or Mac with Java installed. Once opened, you can begin to trade foreign currencies:



Let's keep this window open. As we learn to use the API to send orders to the server, the trades will be reflected in the fxTrade platform.

Exploring the API

The developer page for OANDA is `http://developer.oanda.com`. This page contains a wealth of information related to the development of your trading system in interfacing with the API.

Getting oandapy – the OANDA REST API wrapper

Since we will be using Python for our forex trading platform, let's obtain oandapy, the Python wrapper for the REST API at `https://github.com/oanda/oandapy`. Download the `oandapy.py` file to your working directory.

The use of `oandapy.py` requires the python-requests module. Assuming that you have pip installed, run the following command in the Terminal:

```
$ pip install requests
```

Now we are ready to use Python to interface with OANDA's REST API.

Getting and parsing rates data

We will get the current rates using the oandapy module. Let's begin by defining the `account_id` and `key` variables to store our account identifier and access token respectively:

```
>>> account_id = 6858884
>>> key = "4c7718c7e03d472c2369abf1cb7ceddb-" \
>>>       "142b7d845d68844e853bb95c63f1c8b9"
```

Next, we will create an instance of the `oandapy.API`, as the `oanda` variable with the following code. The `get_prices` method is invoked by passing the `instruments` parameter with a string value of `EUR_USD` so that the rates of the EUR/USD currency pair are fetched:

```
>>> """ Fetch rates """
>>> import oandapy
>>>
>>> oanda = oandapy.API(environment="practice", access_token=key)
>>> response = oanda.get_prices(instruments="EUR_USD")
```

Let's view the returned response data:

```
>>> print response
{u'prices': [{u'ask': 1.0976, u'instrument': u'EUR_USD', u'bid':
1.09744, u'time': u'2015-03-26T02:15:30.015091Z'}]}
```

The returned data is of a dictionary object type with the `prices` key that contains a list of corresponding prices. Let's parse the data of the first item in the list with the `bid`, `ask`, `instrument`, and `time` keys, and assign them to separate variables:

```
>>> prices = response["prices"]
>>> bidding_price = float(prices[0]["bid"])
>>> asking_price = float(prices[0]["ask"])
>>> instrument = prices[0]["instrument"]
>>> time = prices[0]["time"]
```

We can output each variable to ensure that we have parsed the dictionary data correctly:

```
>>> print "[%s] %s bid=%s ask=%s" % (
>>>     time, instrument, bidding_price, asking_price)
[2015-03-26T02:22:54.776348Z] EUR_USD bid=1.09746 ask=1.09762
```

The current bid price and ask price for EUR/USD is 1.09746 and 1.09762 respectively.

Sending an order

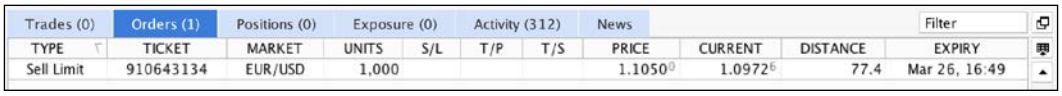
Let's send a limit order to the server with the following code. Note that the expiry time is required for a limit order, stop order, or market-if-touched order. We will send a limit order to sell 1,000 units of EUR/USD at 1.105 with a one-day expiry period:

```
>>> """ Send an order """
>>> from datetime import datetime, timedelta
>>>
>>> # set the trade to expire after one day
>>> trade_expire = datetime.now() + timedelta(days=1)
>>> trade_expire = trade_expire.isoformat("T") + "Z"
>>>
>>> response = oanda.create_order(
>>>     account_id,
```

```
>>>     instrument="EUR_USD",
>>>     units=1000,
>>>     side="sell", # "buy" or "sell"
>>>     type="limit",
>>>     price=1.105,
>>>     expiry=trade_expire)
>>> print response
{u'orderOpened': {u'lowerBound': 0, u'stopLoss': 0, u'takeProfit': 0,
u'upperBound': 0, u'expiry': u'2015-03-26T21:42:28.000000Z',
u'trailingStop': 0, u'units': 1000, u'id': 910641795, u'side':
u'sell'}, u'instrument': u'EUR_USD', u'price': 1.105, u'time':
u'2015-03-26T02:42:28.000000Z'}
```

Valid order types for the `type` parameter are `limit`, `stop`, `marketIfTouched`, and `market`.

Note that the `oanda` and `account_id` variables are reused from the previous section. The response data is of a dictionary object type. When successful, the limit order trade will appear in the **fxTrade Practice** platform under the **Orders** tab, as shown in the following screenshot:



Trades (0)	Orders (1)	Positions (0)	Exposure (0)	Activity (312)	News					Filter
TYPE	TICKET	MARKET	UNITS	S/L	T/P	T/S	PRICE	CURRENT	DISTANCE	EXPIRY
Sell Limit	910643134	EUR/USD	1,000				1.1050[0]	1.0972[6]	77.4	Mar 26, 16:49

Building a trend-following forex trading platform

Suppose from the forex tick data collected, we resample the time series at regular intervals. The average of prices over a reasonably short time period and long time period are calculated. The beta of the price series is taken as the ratio of the short-term average prices to the long-term average prices. In price series, where there is no trend, the ratio is 1—short-term prices are equal to the long-term prices. When prices are on an uptrend, the short-term prices are higher than the long-term average price levels and the beta is more than 1. Conversely, when prices are on a downtrend, the beta is less than 1.

In this section, we will discuss the implementation of a trend-following trading system to buy a position when prices are in an uptrend and sell when prices are going downtrend.

Setting up the main program

Let's create a class named `ForexSystem` that inherits the `oandapy.Streamer` class with the following required variables in the constructor:

```
"""
Implementing the trend-following algorithm
for trading foreign currencies
"""
import oandapy
from datetime import datetime
import pandas as pd


class ForexSystem(oandapy.Streamer):
    def __init__(self, *args, **kwargs):
        oandapy.Streamer.__init__(self, *args, **kwargs)
        self.oanda = oandapy.API(kwargs["environment"],
                                 kwargs["access_token"])

        self.instrument = None
        self.account_id = None
        self.qty = 0
        self.resample_interval = '10s'
        self.mean_period_short = 5
        self.mean_period_long = 20
        self.buy_threshold = 1.0
        self.sell_threshold = 1.0

        self.prices = pd.DataFrame()
        self.beta = 0
        self.is_position_opened = False
        self.opening_price = 0
        self.executed_price = 0
        self.unrealized_pnl = 0
        self.realized_pnl = 0
        self.position = 0
        self.dt_format = "%Y-%m-%dT%H:%M:%S.%fZ"
```

We will create a method called `begin` in the `ForexSystem` class as the starting point of the program. Note that invoking `self.start` will begin streaming rates data:

```
def begin(self, **params):
    self.instrument = params["instruments"]
    self.account_id = params["accountId"]
    self.qty = params["qty"]
    self.resample_interval = params["resample_interval"]
    self.mean_period_short = params["mean_period_short"]
    self.mean_period_long = params["mean_period_long"]
    self.buy_threshold = params["buy_threshold"]
    self.sell_threshold = params["sell_threshold"]

    self.start(**params)  # Start streaming prices
```

Handling events

The `on_success` method is inherited from the `oandapy.Streamer` class that will handle the incoming rates data into our system. In the following code, we will parse this data into their respective variables for use in the `tick_event` method:

```
def on_success(self, data):
    time, symbol, bid, ask = self.parse_tick_data(
        data["tick"])
    self.tick_event(time, symbol, bid, ask)
```

Implementing the trend-following algorithm

The `tick_event` method will process the tick data information by resampling the time series to calculate the beta. Note that we will store the mid-price of the bid and ask price. The beta will be used in the `perform_trade_logic` method to determine whether to open or close our position. The status of the system is printed on every method call:

```
def tick_event(self, time, symbol, bid, ask):
    midprice = (ask+bid)/2.
    self.prices.loc[time, symbol] = midprice

    resampled_prices = self.prices.resample(
        self.resample_interval,
        how='last',
        fill_method="ffill")

    mean_short = resampled_prices.tail(
```

```
            self.mean_period_short).mean()[0]
        mean_long = resampled_prices.tail(
            self.mean_period_long).mean()[0]
        self.beta = mean_short / mean_long

        self.perform_trade_logic(self.beta)
        self.calculate_unrealized_pnl(bid, ask)
        self.print_status()
```

In the following `perform_trade_logic` method, a buy signal indicates that a new long position is to be opened, or an existing short position is to be closed, by sending a buy market order. Conversely, a sell signal indicates that a new short position is to be opened, or an existing long position is to be closed, by sending a sell market order:

```
def perform_trade_logic(self, beta):

    if beta > self.buy_threshold:
        if not self.is_position_opened \
                or self.position < 0:
            self.check_and_send_order(True)

    elif beta < self.sell_threshold:
        if not self.is_position_opened \
                or self.position > 0:
            self.check_and_send_order(False)
```

The full source code for the `ForexSystem` class is available at the Packt Publishing website.

Tracking our positions

The `print_status` method displays the time, currency pair, position, beta, and profits and losses of our position during the running lifetime of the system with the following code:

```
def print_status(self):
    print "[%s] %s pos=%s beta=%s RPnL=%s UPnL=%s" % (
        datetime.now().time(),
        self.instrument,
        self.position,
        round(self.beta, 5),
        self.realized_pnl,
        self.unrealized_pnl)
```

Let's start running our algorithmic trading system with the following code:

```
if __name__ == "__main__":
    key = "4c7718c7e03d472c2369abf1cb7ceddb-" \
          "142b7d845d68844e853bb95c63f1c8b91"
    account_id = 6858884
    system = ForexSystem(environment="practice", access_token=key)
    system.begin(accountId=account_id,
                 instruments="EUR_USD",
                 qty=1000,
                 resample_interval="10s",
                 mean_period_short=5,
                 mean_period_long=20,
                 buy_threshold=1.,
                 sell_threshold=1.)
```

Here, we are specifying the system to trade 1,000 units of EUR/USD each time. The resampling period of the time series is 10-second intervals. The short-term averaging period is defined to be the recent five periods or fifty seconds. The long-term averaging period is defined to be the recent twenty periods or two hundred seconds. When the beta exceeds the buy threshold value of 1, the system will enter into a long position of 1,000 units. Otherwise, when the beta falls below the sell threshold of 1, the system will enter into a short position of 1,000 units.

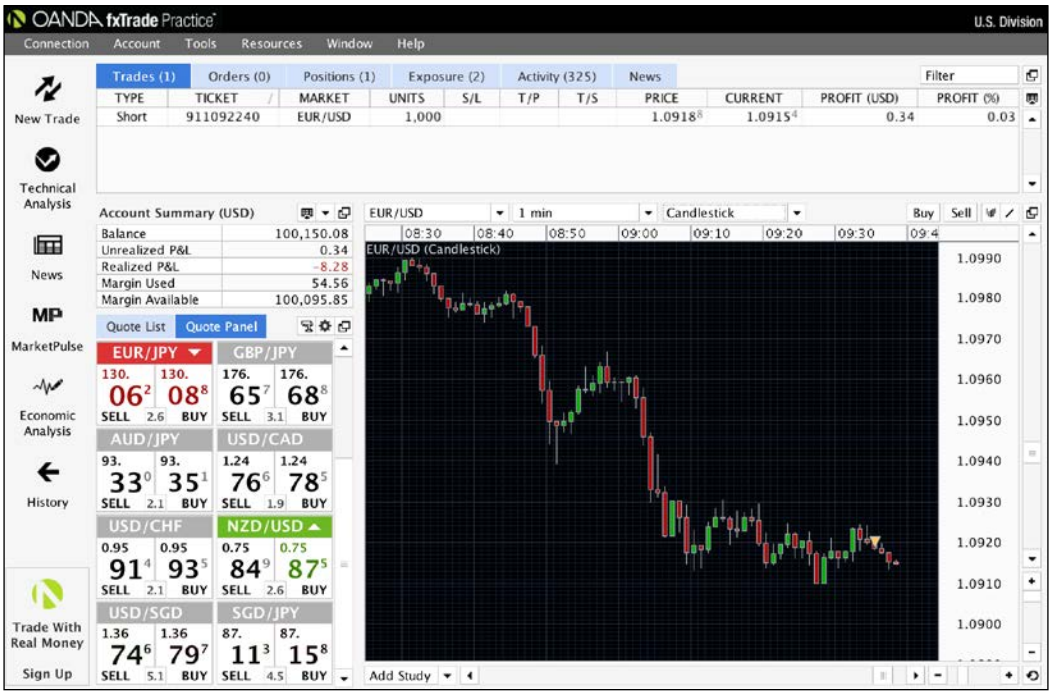
The first few lines of the output should give us the following result:

```
[09:31:59.067633] EUR_USD pos=0 beta=1.0 RPnL=0 UPnL=0
[09:31:59.163893] EUR_USD pos=0 beta=1.0 RPnL=0 UPnL=0
[09:32:00.233068] EUR_USD pos=0 beta=1.0 RPnL=0 UPnL=0
```

Suppose after some time, the value of our beta decreases. Our trading system will follow the trend by placing a sell market order of 1,000 units of EUR/USD. Our output status will update with the following information:

```
[09:35:42.305521] EUR_USD pos=0 beta=1.0 RPnL=0 UPnL=0
Placed order sell 1000 EUR_USD at market.
[09:35:42.765773] EUR_USD pos=-1000 beta=0.99999 RPnL=0 UPnL=-0.14
[09:35:48.434842] EUR_USD pos=-1000 beta=0.99999 RPnL=0 UPnL=-0.11
…
[09:38:28.864373] EUR_USD pos=-1000 beta=0.99984 RPnL=0 UPnL=0.32
[09:38:29.096078] EUR_USD pos=-1000 beta=0.99984 RPnL=0 UPnL=0.31
```

In the **fxTrade Practice** platform, we should be able to view our trade:



Our trading system will run indefinitely until we terminate the process with a *Ctrl* + *Z* or something similar.

Although our trend-following trading system seems to be doing reasonable well, however, as discussed in the previous sections, more improvements are required to develop a profitable and robust trading strategy.

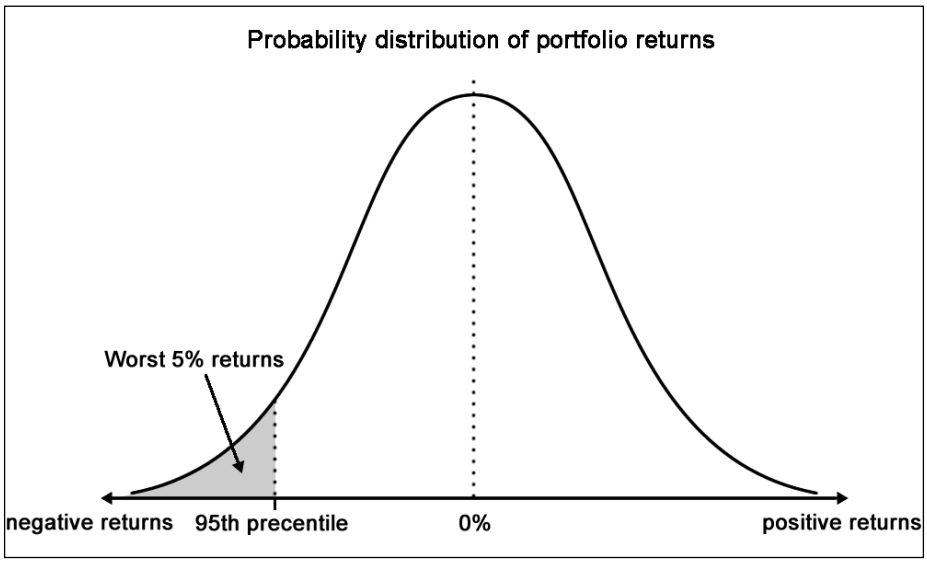
In the next section, we will discuss risk management for our trading systems.

VaR for risk management

As soon as we open a position in the market, we are exposed to various types of risks, such as volatility risk and credit risk. To preserve our trading capital as much as possible, it is important to incorporate some form of risk management measures to our trading system.

Perhaps the most common measure of risk used in the financial industry is the VaR technique. It is designed to simply answer: what is the worst expected amount of loss, given a specific probability level, say 95 percent, over a certain period of time? The beauty of VaR is that it can be applied to multiple levels, from position-specific micro level to portfolio-based macro level. For example, a VaR of $1 million with a 95 percent confidence level for a one-day time horizon states that on average, only 1 day out of 20 would you expect to lose more than $1 million due to market movements.

The following figure illustrates a normally distributed portfolio returns with a mean of 0 percent, where VaR is the loss corresponding to the 95th percentile of the distribution of portfolio returns:



Suppose we have $100 million under management at a fund claiming to have the same risk as an S&P 500 index fund, with an expected return of 9 percent and a standard deviation of 20 percent. To calculate the daily VaR at the 5 percent risk level or 95 percent confidence using the variance-covariance method, we will use the following formulas:

$$daily\ volatility, \sigma = \frac{20\%}{\sqrt{252}} = 1.26\%$$

$$daily\ expected\ return, u = \frac{9\%}{252} = 0.036\%$$

$$VaR = P - P\left(N^{-1}(\alpha, u, \sigma) + 1\right) = \$2,036,606.50$$

Here, P is the value of the portfolio, and $N^{-1}(\alpha,u,\sigma)$ is the inverse normal probability distribution with risk level of α, mean of u, and standard deviation of σ. The number of trading days per year is assumed to be 252. It turns out that the VaR is $2,036,606.50.

However, the use of VaR is not without its flaws. It does not take into account the probability of the loss for extreme events happening on the far ends of the tails on the normal distribution curve. The magnitude of the loss beyond a certain VaR level would be difficult to estimate as well. The VaR that we investigated uses historical data and an assumed constant volatility level. Such measures are not indicative of our future performance.

Let's take a practical approach to calculate the daily VaR of a set of stock prices from Yahoo! Finance. We will investigate the AAPL stock:

```
import datetime as dt
import numpy as np
import pandas.io.data as rda
from scipy.stats import norm

def calculate_daily_VaR(P, prob, mean, sigma,
                        days_per_year=252.):
    min_ret = norm.ppf(1-prob,
                       mean/days_per_year,
                       sigma/np.sqrt(days_per_year))
    return P - P*(min_ret+1)

if __name__ == "__main__":
    start = dt.datetime(2013, 12, 1)
    end = dt.datetime(2014, 12, 1)

    prices = rda.DataReader("AAPL", "yahoo", start, end)
    returns = prices["Adj Close"].pct_change().dropna()

    portvolio_value = 100000000.00
    confidence = 0.95
    mu = np.mean(returns)
    sigma = np.std(returns)

    VaR = calculate_daily_VaR(portvolio_value, confidence,
                              mu, sigma)
    print "Value-at-Risk:", round(VaR, 2)
```

The `calculate_daily_VaR` function performs the daily VaR calculation, assuming 252 trading days per year. The `mean` and `sigma` variables are annualized values of the average and standard deviation of daily stock returns respectively. The `norm.ppf` method of the `scipy.stats` module performs the inverse of the normal probability with a risk level of `1-prob`, where `prob` is the confidence value of interest.

The `pandas.io.data.DataReader` function is a nifty feature of pandas that provides remote data access to certain data sources, including the following:

- Yahoo! Finance
- Google Finance
- St. Louis Fed (FRED)
- Kenneth French's data library
- World Bank
- Google Analytics

We chose Yahoo! Finance with the `yahoo` value encapsulated in string quotes in the function arguments, along with the start and end dates of the data. The `Adj Close` column of the returned `DataFrame` object is used to compute the daily percentage price changes. Finally, we will call the `calculate_daily_VaR` function to compute our daily VaR:

```
Value-at-Risk: 138755.57
```

The daily VaR for the stock AAPL with 95 percent confidence is $138,755.57.

Summary

In this chapter, we were introduced to the evolution of trading from the pits to the electronic trading platform, and learned how algorithmic trading came about. We looked at some brokers offering API access to their trading service offering. To help us get started on our journey in developing an algorithmic trading system, we used the TWS of IB and the IbPy Python module.

In our first trading program, we successfully sent an order to our broker through the TWS API using a demonstration account. Next, we developed a simple algorithmic trading system. We started by requesting the market data and account updates from the server. With the captured real-time information, we implemented a mean-reverting algorithm to trade the markets. Since this trading system uses only one indicator, more work would be required to build a robust, reliable, and profitable trading system.

We also discussed currency trading with the OANDA REST API with the help of the oandapy Python module. After setting up our account for API access, our first step to explore the OANDA API is to fetch rates for a single currency pair and send a limit order to the server. Using the fxTrade Practice platform, we can track our current trades, orders, and positions. Next, we developed a trend-following algorithm to trade the EUR/USD currency pair with the use of streaming rates and market orders.

One critical aspect of trading is to manage risk effectively. In the financial industry, the VaR is the most common technique used to measure risk. Using Python, we took a practical approach to calculate the daily VaR of a set of stock prices from Yahoo! Finance.

Once we have built a working algorithmic trading system, we can explore the other ways to measure the performance of our trading strategy. One of these areas is backtesting. We will discuss this topic in the next chapter.

9
Backtesting

A **backtest** is a simulation of a model-driven investment strategy's response to historical data. While working on designing and developing a backtest, it would be helpful to think in terms of the concept of creating video games.

In this chapter, we will design and implement an event-driven backtesting system using object-oriented design. We can then plot our resulting profits and losses onto a graph to help us visualize the performance of our trading strategy. Is this sufficient to deduce a good model?

There are many concerns to be addressed in backtesting, for example, the effects of transaction costs, execution latency of orders, access to detailed transactions, and quality of historical data. Notwithstanding these factors, the primary goal of creating a backtesting system is to test a model as accurately as possible.

Backtesting involves a lot of research that merits its own literature. We will briefly cover some thoughts that you might want to consider when implementing a backtest. Typically, a number of algorithms are employed in backtesting. We will briefly discuss some of these: k-means clustering, k-nearest neighbor, classification and regression tree, 2k factorial design, and genetic algorithm.

In this chapter, we will cover the following topics:

- An introduction to backtesting
- Concerns in backtesting
- Concept of an event-driven backtesting system
- Designing and implementing a backtesting system
- Creating the TickData, MarketData, MarketDataSource, and Order class
- Creating the Position, Strategy, and MeanRevertingStrategy class
- Creating and running the Backtester class
- Ten considerations for a backtesting model
- Discussion of algorithms in backtesting

An introduction to backtesting

A backtest is a simulation of a model-driven investment strategy's response to historical data. The purpose of performing experiments with backtests is to make discoveries about a process or system. Using historical data, you can save time in testing an investment strategy for the period forward. It helps you test an investment theory based on the movements of the tested period. It is also used to both evaluate and calibrate an investment model.

Creating a model is only the first step. The investment strategy will typically employ the model to help you drive simulated trading decisions and compute various factors related to either risk or return. These factors are typically used together to find a combination that is predictive of return.

Concerns in backtesting

However, there are many concerns to be addressed in backtesting. A backtest can never exactly replicate the performance of an investment strategy in an actual trading environment. The quality of the historical data is of question, since it is subjected to outliers by third-party data vendors. Look-ahead bias takes many forms. For example, listed companies may split, merge, or delist, resulting in substantial changes to its stock price. For strategies based on information from the order book, the market microstructure is extremely difficult to simulate realistically, since it represents the collective visible demand and supply in continuous time. This demand and supply are in turn affected by news events around the world. Icebergs and resting orders are some hidden elements of the market that could affect the structure once activated. Other factors to be considered are transaction costs, execution latency of orders, and access to detailed transactions from backtesting. Notwithstanding these factors, the primary goal of creating a backtesting system is thus to test a model as accurately as possible.

Look-ahead bias is the use of available future data during the period it is being analyzed, resulting in inaccurate results in the simulation or study. It is vital to use information that would be only available during the period of study.

In finance, iceberg orders are large orders that are broken up into several small orders. Only a small portion of the order is visible to the public—just like the "tip of the iceberg"—while the mass of the actual order is hidden.

A resting order is an order whose price is away from the market and is waiting to be executed.

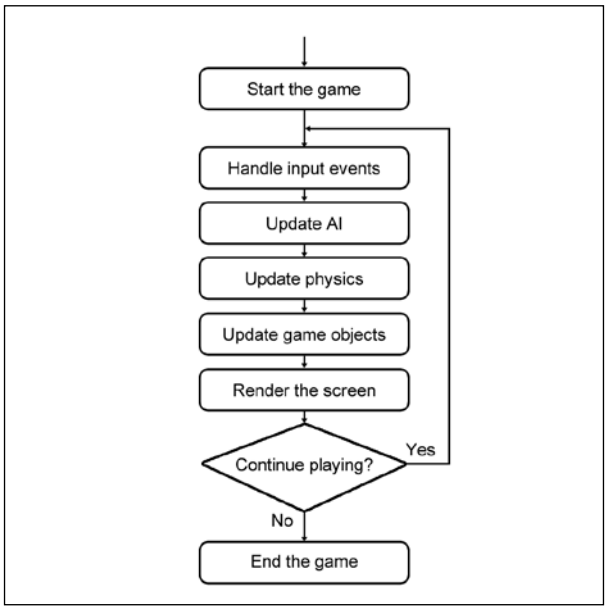
Concept of an event-driven backtesting system

While working on designing and developing a backtest, it would be helpful to think in terms of the concept of creating video games. After all, we are trying to create a simulated market pricing and ordering environment, very much akin to creating a virtual gaming world. Trading can also be regarded as a thrilling game of buying low and selling high.

In a virtual trading environment, components are needed for the simulation of price feeds, the order matching engine, the order book management as well as for account and position updates. To achieve these functionalities, we can explore the concept of an event-driven backtesting system.

Let's start by understanding the concept of an event-driven programming paradigm used throughout the game development process. A system typically receives events as its inputs. It might be a keystroke entered by a user or a mouse movement. Other events could be messages that are generated by another system, a process, or a sensor to notify the host system of an incoming event.

The following flowchart illustrates the stages involved in a game engine system:



Let's take a look at a pseudocode implementation of a main game engine loop:

```
while is_main_loop: # Main game engine loop
  handle_input_events()
  update_AI()
  update_physics()
  update_game_objects()
  render_screen()
  sleep(1/60) # Assuming a 60 frames-per-second video game rate
```

The core functions within the main game engine loop may process generated system events, as in the case of the `handle_input_events` function to handle the keyboard events:

```
def handle_input_events()
  event = get_latest_event()

  if event.type == "UP_KEY_PRESS":
    move_player_up()
  elif event.type == "DOWN_KEY_PRESS":
    move_player_down()
```

Using an event-driven system, such as the preceding example, helps us achieve code modularity and reusability by being able to swap and use similar events from different system components. The use of object-oriented programming is further enforced, where classes define objects in a game. These features are particularly useful for interfacing with different market data sources, multiple trading algorithms, and runtime environments when designing our trading platform. The simulated trading environment is close to being a realistic one and helps us prevent look-ahead bias.

Designing and implementing a backtesting system

Now that we have an idea of designing a video game for creating a backtesting trading system, we can begin our object-oriented approach by first defining the required classes for the various components in our trading system.

We are interested in implementing a simple backtesting system to test a mean-reverting strategy. Using the daily historical prices from Google Finance, we will take the closing price of each day to compute the volatility of price returns for a particular stock, using the ticker symbol AAPL as an example. We want to test a theory that if the standard deviation of returns for an elapsed number of days is far from the mean of zero by a particular threshold, a buy or sell signal is generated. When such a signal is indeed generated, a market order is sent to the exchange to be executed at the opening price of the next trading day.

As soon as we open a position, we would like to track our unrealized and realized profits till date. Our open position can be closed when an opposing signal is generated. On completion of the backtest, we will plot our profits and losses to see how well our strategy holds.

Does our theory sound like a viable trading strategy? Well, let's find out! The following sections explain the classes that will be used for implementing a backtesting system.

The TickData class

The `TickData` class represents a single unit of data received from a market data source. In this example, we are interested in just the stock symbol, the timestamp of the data, the opening price, and the last price:

```
""" Store a single unit of data """
class TickData:
    def __init__(self, symbol, timestamp,
                 last_price=0, total_volume=0):
        self.symbol = symbol
        self.timestamp = timestamp
        self.open_price = 0
        self.last_price = last_price
        self.total_volume = total_volume
```

Detailed descriptions of a single unit of tick data, such as the total volume, bid price, ask price, or last volume can be added as our system evolves.

The MarketData class

An instance of this class is used throughout the system to store and retrieve prices by the various components. Essentially, a container is used to store the last tick data. Additional helper functions are included to provide easy reference to the required information:

```
class MarketData:
    def __init__(self):
        self.__recent_ticks__ = dict()

    def add_last_price(self, time, symbol, price, volume):
        tick_data = TickData(symbol, time, price, volume)
        self.__recent_ticks__[symbol] = tick_data

    def add_open_price(self, time, symbol, price):
        tick_data = self.get_existing_tick_data(symbol, time)
        tick_data.open_price = price

    def get_existing_tick_data(self, symbol, time):
        if not symbol in self.__recent_ticks__:
            tick_data = TickData(symbol, time)
            self.__recent_ticks__[symbol] = tick_data

        return self.__recent_ticks__[symbol]

    def get_last_price(self, symbol):
        return self.__recent_ticks__[symbol].last_price

    def get_open_price(self, symbol):
        return self.__recent_ticks__[symbol].open_price

    def get_timestamp(self, symbol):
        return self.__recent_ticks__[symbol].timestamp
```

The MarketDataSource class

The `MarketDataSource` class helps us fetch historical data from an external source, such as Google Finance or Yahoo! Finance. The required parameter values, such as `start`, `end`, `ticker`, and `source` are provided from the host component of this class, which we will discuss later. After saving the opening and closing prices of each day, the `event_tick` variable that represents a function handled by the host component will be invoked on every tick event. Notice that we are using the `DataReader` function of pandas to retrieve historical prices. The acceptable parameters are `yahoo` for Yahoo! Finance data source and `google` for Google Finance data source:

```
import pandas.io.data as web

""" Download prices from an external data source """
```

```
class MarketDataSource:
    def __init__(self):
        self.event_tick = None
        self.ticker, self.source = None, None
        self.start, self.end = None, None
        self.md = MarketData()

    def start_market_simulation(self):
        data = web.DataReader(self.ticker, self.source,
                              self.start, self.end)

        for time, row in data.iterrows():
            self.md.add_last_price(time, self.ticker,
                                   row["Close"], row["Volume"])
            self.md.add_open_price(time, self.ticker, row["Open"])

            if not self.event_tick is None:
                self.event_tick(self.md)
```

The Order class

The `Order` class represents a single order sent by the strategy to the server. Each order contains a timestamp, the symbol, quantity, price, and the size of the order. In this example, we are using market orders only. Other order types, such as limit and stop orders, can be further implemented if desired. Once an order is filled, the order is further updated with the filled time, quantity, and price:

```
class Order:
    def __init__(self, timestamp, symbol, qty, is_buy,
                 is_market_order, price=0):
        self.timestamp = timestamp
        self.symbol = symbol
        self.qty = qty
        self.price = price
        self.is_buy = is_buy
        self.is_market_order = is_market_order
        self.is_filled = False
        self.filled_price = 0
        self.filled_time = None
        self.filled_qty = 0
```

The Position class

The `Position` class helps us keep track of our current market position and account balance. Note that the `position_value` variable starts with a value of zero. When stocks are bought, the value of the securities is debited from this account. When stocks are sold, the value of the securities is credited into this account:

```
class Position:
    def __init__(self):
        self.symbol = None
        self.buys, self.sells, self.net = 0, 0, 0
        self.realized_pnl = 0
        self.unrealized_pnl = 0
        self.position_value = 0

    def event_fill(self, timestamp, is_buy, qty, price):
        if is_buy:
            self.buys += qty
        else:
            self.sells += qty

        self.net = self.buys - self.sells
        changed_value = qty * price * (-1 if is_buy else 1)
        self.position_value += changed_value

        if self.net == 0:
            self.realized_pnl = self.position_value

    def update_unrealized_pnl(self, price):
        if self.net == 0:
            self.unrealized_pnl = 0
        else:
            self.unrealized_pnl = price * self.net + \
                                  self.position_value

        return self.unrealized_pnl
```

The Strategy class

The `Strategy` class is the base class for all other strategy implementations. The `event_tick` method is called when new market tick data arrives. The `event_order` method is called whenever there are order updates. The `event_position` method is called whenever there are updates to our positions. The `send_market_order` method is called when the implementing strategy sends a market order to the host component to be routed to the server for execution:

```
""" Base strategy for implementation """
class Strategy:
    def __init__(self):
        self.event_sendorder = None

    def event_tick(self, market_data):
        pass

    def event_order(self, order):
        pass

    def event_position(self, positions):
        pass

    def send_market_order(self, symbol, qty, is_buy, timestamp):
        if not self.event_sendorder is None:
            order = Order(timestamp, symbol, qty, is_buy, True)
            self.event_sendorder(order)
```

The MeanRevertingStrategy class

In this example, we are implementing a mean-reverting strategy with the `MeanRevertingStrategy` class that inherits the `Strategy` class. We will use the stock symbol AAPL.

The `event_position` method is overridden and updates the state of the strategy to indicate a long or a short on every change in position. Knowing the current state of the strategy prevents us from adding on to our positions and entering more orders than intended.

The `event_tick` method is overridden to perform the trade logic decision on every incoming tick data, which is stored as a pandas `DataFrame` object, to calculate the strategy parameters. The `lookback_intervals` variable defines a maximum of 20 days of historical prices to store.

The `calculate_z_score` method implements our mean-reverting calculations. The daily percentage change of close prices over the previous day is computed. The `dropna` function removes any empty values from the result. The returns are then Z-scored, such as:

$$Z - score = \frac{x - \mu}{\sigma}$$

Here, x is the most recent return, μ is the mean of returns, and σ is the standard deviation of returns. A `z_score` value of 0 indicates that the score is the same as the mean. When the value of `z_score` reaches 1.5 or -1.5, as defined by the `sell_threshold` and `buy_threshold` variables respectively, this could indicate a strong sell or buy signal, since the Z-score for the following periods is expected to revert back to the mean of zero. When a signal is generated it can be used to either open a position or to close an existing position:

```
"""
Implementation of a mean-reverting strategy
based on the Strategy class
"""
import pandas as pd

class MeanRevertingStrategy(Strategy):
    def __init__(self, symbol,
                 lookback_intervals=20,
                 buy_threshold=-1.5,
                 sell_threshold=1.5):
        Strategy.__init__(self)
        self.symbol = symbol
        self.lookback_intervals = lookback_intervals
        self.buy_threshold = buy_threshold
        self.sell_threshold = sell_threshold
        self.prices = pd.DataFrame()
        self.is_long, self.is_short = False, False

    def event_position(self, positions):
        if self.symbol in positions:
            position = positions[self.symbol]
            self.is_long = True if position.net > 0 else False
            self.is_short = True if position.net < 0 else False

    def event_tick(self, market_data):
        self.store_prices(market_data)
```

```
        if len(self.prices) < self.lookback_intervals:
            return

        signal_value = self.calculate_z_score()
        timestamp = market_data.get_timestamp(self.symbol)

        if signal_value < self.buy_threshold:
            self.on_buy_signal(timestamp)
        elif signal_value > self.sell_threshold:
            self.on_sell_signal(timestamp)

    def store_prices(self, market_data):
        timestamp = market_data.get_timestamp(self.symbol)
        self.prices.loc[timestamp, "close"] = \
            market_data.get_last_price(self.symbol)
        self.prices.loc[timestamp, "open"] = \
            market_data.get_open_price(self.symbol)

    def calculate_z_score(self):
        self.prices = self.prices[-self.lookback_intervals:]
        returns = self.prices["close"].pct_change().dropna()
        z_score = ((returns-returns.mean())/returns.std())[-1]
        return z_score

    def on_buy_signal(self, timestamp):
        if not self.is_long:
            self.send_market_order(self.symbol, 100,
                                   True, timestamp)

    def on_sell_signal(self, timestamp):
        if not self.is_short:
            self.send_market_order(self.symbol, 100,
                                   False, timestamp)
```

The Backtester class

After defining all of our core components, we are now ready to implement the backtesting engine as the `Backtester` class.

The `start_backtest` method initializes our strategy, defines the order handler for this strategy with the `evthandler_order` method, sets up and runs the market data source function. When data is received from the market data source function, the function `evthandler_tick` method handles each incoming tick data and passes them to our strategy.

Thereafter, the `match_order_book` method, in conjunction with the `is_order_unmatched` method, is called to make an attempt to match any outstanding orders in our system, given the current market prices. The `is_order_unmatched` method returns `True` when no order is filled, or `False` otherwise. On filling an order, it calls the `update_filled_position` method for further processing. This includes updating the position values, notifying the `Strategy` object of a position update, and keeping track of our profits and losses. The `is_order_unmatched` method also notifies the `Strategy` object of an order update event when an order is filled.

Lastly, the position updates are printed to the console to help us keep track of our account status. This main loop of the backtesting engine continues until the last tick is available from the source of the market data. The full implementation of the `Backtester` class is given as follows:

```
import datetime as dt
import pandas as pd

class Backtester:
    def __init__(self, symbol, start_date, end_date,
                 data_source="google"):
        self.target_symbol = symbol
        self.data_source = data_source
        self.start_dt = start_date
        self.end_dt = end_date
        self.strategy = None
        self.unfilled_orders = []
        self.positions = dict()
        self.current_prices = None
        self.rpnl, self.upnl = pd.DataFrame(), pd.DataFrame()

    def get_timestamp(self):
        return self.current_prices.get_timestamp(
            self.target_symbol)

    def get_trade_date(self):
        timestamp = self.get_timestamp()
        return timestamp.strftime("%Y-%m-%d")

    def update_filled_position(self, symbol, qty, is_buy,
                               price, timestamp):
        position = self.get_position(symbol)
        position.event_fill(timestamp, is_buy, qty, price)
        self.strategy.event_position(self.positions)
        self.rpnl.loc[timestamp, "rpnl"] = position.realized_pnl
```

```
        print self.get_trade_date(), \
            "Filled:", "BUY" if is_buy else "SELL", \
            qty, symbol, "at", price

    def get_position(self, symbol):
        if symbol not in self.positions:
            position = Position()
            position.symbol = symbol
            self.positions[symbol] = position

        return self.positions[symbol]

    def evthandler_order(self, order):
        self.unfilled_orders.append(order)

        print self.get_trade_date(), \
            "Received order:", \
            "BUY" if order.is_buy else "SELL", order.qty, \
             order.symbol

    def match_order_book(self, prices):
        if len(self.unfilled_orders) > 0:
            self.unfilled_orders = \
                [order for order in self.unfilled_orders
                 if self.is_order_unmatched(order, prices)]

    def is_order_unmatched(self, order, prices):
        symbol = order.symbol
        timestamp = prices.get_timestamp(symbol)

        if order.is_market_order and timestamp > order.timestamp:
            # Order is matched and filled.
            order.is_filled = True
            open_price = prices.get_open_price(symbol)
            order.filled_timestamp = timestamp
            order.filled_price = open_price
            self.update_filled_position(symbol,
                                        order.qty,
                                        order.is_buy,
                                        open_price,
                                        timestamp)
            self.strategy.event_order(order)
            return False
```

```
            return True

        def print_position_status(self, symbol, prices):
            if symbol in self.positions:
                position = self.positions[symbol]
                close_price = prices.get_last_price(symbol)
                position.update_unrealized_pnl(close_price)
                self.upnl.loc[self.get_timestamp(), "upnl"] = \
                    position.unrealized_pnl

                print self.get_trade_date(), \
                    "Net:", position.net, \
                    "Value:", position.position_value, \
                    "UPnL:", position.unrealized_pnl, \
                    "RPnL:", position.realized_pnl

        def evthandler_tick(self, prices):
            self.current_prices = prices
            self.strategy.event_tick(prices)
            self.match_order_book(prices)
            self.print_position_status(self.target_symbol, prices)

        def start_backtest(self):
            self.strategy = MeanRevertingStrategy(self.target_symbol)
            self.strategy.event_sendorder = self.evthandler_order

            mds = MarketDataSource()
            mds.event_tick = self.evthandler_tick
            mds.ticker = self.target_symbol
            mds.source = self.data_source
            mds.start, mds.end = self.start_dt, self.end_dt

            print "Backtesting started..."
            mds.start_market_simulation()
            print "Completed."
```

Running our backtesting system

To run our backtester, simply create an instance of the class with the required parameters. Here, we defined the ticker symbol `AAPL` for the period January 1, 2014 to December 31, 2014. By default, our target market data source is defined as `google`. Then, we will call the `start_backtest` method:

```
>>> backtester = Backtester("AAPL",
...                                   dt.datetime(2014, 1, 1),
```

```
...                                          dt.datetime(2014, 12, 31))
>>> backtester.start_backtest()
```

The output will begin to run like this:

```
Backtesting started...
2014-02-27 Received order: SELL 100 AAPL
2014-02-28 Filled: SELL 100 AAPL at 75.58
2014-02-28 Net: -100 Value: 7558.0 UPnL: 40.0 RPnL: 0
2014-03-03 Net: -100 Value: 7558.0 UPnL: 19.0 RPnL: 0
2014-03-04 Net: -100 Value: 7558.0 UPnL: -31.0 RPnL: 0
…
```

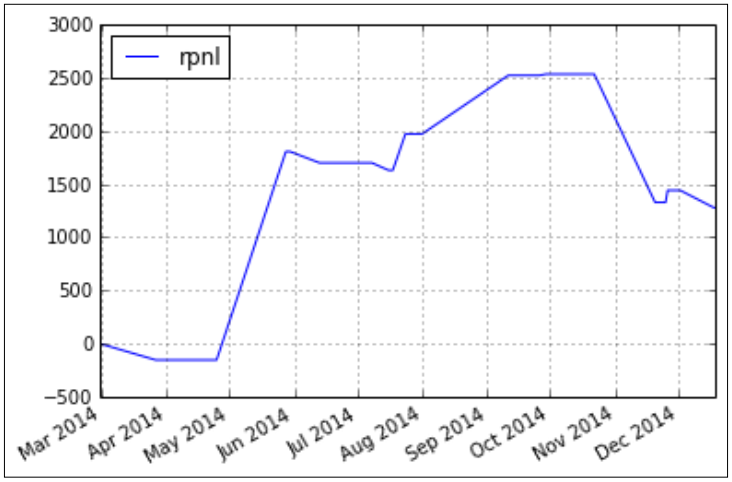
Almost a year's worth of daily information will be printed onto the console. The output will end with something like this:

```
…
014-12-29 Net: -100 Value: 12504.0 UPnL: 1113.0 RPnL: 1278.0
2014-12-30 Net: -100 Value: 12504.0 UPnL: 1252.0 RPnL: 1278.0
2014-12-31 Net: -100 Value: 12504.0 UPnL: 1466.0 RPnL: 1278.0
Completed.
```

In the `MeanRevertingStrategy` class, we trade shares of AAPL in quantities of 100. Note that when the backtest is completed, we still have an outstanding short position of 100 shares. Our realized profit and loss is $1,278, while the unrealized profit from the short position is $1,466.

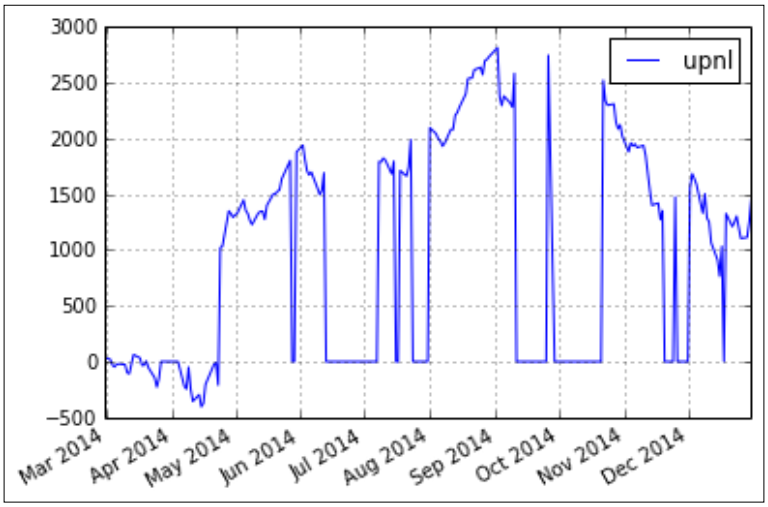
Since we store the daily realized and unrealized profits and losses into a pandas `DataFrame` object, named `rpnl` and `upnl` respectively, we can plot the results to visualize the returns from our strategy:

```
>>> import matplotlib.pyplot as plt
>>> backtester.rpnl.plot()
>>> plt.show()
```




```
>>> backtester.upnl.plot()
>>> plt.show()
```

The following is the output for the preceding commands:



Improving your backtesting system

In this section, we looked at creating a simple backtesting system based on daily closing prices for a mean-reverting strategy. There are several areas of considerations to make such a backtesting model more realistic. Are historical daily prices sufficient to test our model? Should intra-day limit orders be used instead? Our account value started from zero; how can we reflect our capital requirements accurately? Are we able to borrow shares for shorting?

Since we took an object-oriented approach to create a backtesting system, how easy would it be to integrate other components in future? A trading system could accept more than one source of market data. We could also create components that allow us to deploy our system to the product environment.

The list of concerns mentioned are not exhaustive. To guide us in implementing a robust backtesting model, the next section spells out ten considerations in the design of such a system.

Ten considerations for a backtesting model

In the previous section, we performed one replication of a backtest. Our result looks pretty optimistic. However, is this sufficient to deduce that this is a good model? The truth is that backtesting involves a lot of research that stems a literature on its own. The following list briefly covers some thoughts that you might want to consider when implementing your backtests.

Resources restricting your model

The resources that are available to your backtesting system limits how well you can implement your backtest. A financial model that generates signals using only the last closing price needs a set of historical data of closing prices. A trading system that requires reading from the order book requires all levels of the order book data to be available on every tick. This adds up the storage complexity. Other resources, such as exchange data, estimation techniques, and computer resources pose a limitation on the nature of model that can be used.

Criteria of evaluation of the model

How can we conclude that a model is good? Some factors of consideration are Sharpe ratios, hit ratios, average rate of return, VaR statistics, as well as the minimum and maximum drawdown encountered. How can a combination of such factors balance so that a model is usable? How much can the maximum drawdown be tolerated in achieving a high Sharpe ratio?

Estimating the quality of backtest parameters

Using a variety of parameters on a model typically gives us varied results. From multiple models, we can obtain additional sets of data for each model. Can the parameters from the model with the best performance be trustworthy? Using methods such as model averaging can help us correct optimistic estimates.

The model averaging technique is the average fit for a number of models as opposed to using a single best model.

Be prepared to face model risk

Perhaps after extensive backtesting, you may find yourself with a good quality model. How long is it going to stay that way? In model risk, the market structure or the model parameters may change with time, or a regime change may cause the functional form of your model to change abruptly. By then, you could even be uncertain that your model is correct. A solution in addressing model risk is to use model averaging.

Performance of a backtest with in-sample data

Backtesting helps us perform extensive parameter searches that optimize the results of a model. This exploits the true as well as the idiosyncratic aspects of the sample data. Also, historical data can never mimic the way that entire data comes from live markets. These optimized results will always produce an optimistic assessment of the model and the strategy used.

Addressing common pitfalls in backtesting

The most common error made in backtesting is look-ahead bias, and it comes in many forms. For example, parameter estimates may be derived from the entire period of the sample data, which constitute to using information from the future. Statistical estimates like these and model selection should be estimated sequentially, which could actually be difficult to do so.

Errors in data come in all forms, from hardware, software, and human errors that could occur while routed by data distribution vendors. Listed companies may split, merge, or delist, resulting in substantial changes to its stock price. These actions could lead to survivorship bias in our models. Failure to clean data properly will give undue influence to idiosyncratic aspects of data, and thus affect the model parameters.

Survivorship bias is the logical error of concentrating on results that have survived some past selection process. For example, a stock market index may report a strong performance even in bad times because poor performing stocks are dropped from its component weightage, resulting in an overestimation of past returns.

Failure to use shrinkage estimators or model averaging could report results containing extreme values, making it difficult for comparison and evaluation.

In statistics, a shrinkage estimator is used as an alternative to an ordinary least squares estimator to produce the smallest mean squared error. They can be used to shrink raw estimates from the model output towards zero or an other fixed constant value.

Have a common sense idea of your model

Often, common sense could be lacking in our models. We may attempt to explain a trendless variable with a trended variable or infer causation from correlation. Can logarithmic values be used when the context does or does not require it?

Understanding the context for the model

Having a common sense idea of a model is barely sufficient. A good model takes into account the history, personnel involved, operating constraints, common peculiarities, and all the understanding for the rationale of the model. Are commodity prices following seasonal movements? How was the data gathered? Are the formulas used in the computation of variables reliable? These questions can help us determine the causes, should things go wrong.

Make sure you have the right data

Not many of us have access to tick-level data. Low-resolution tick data may miss out on detailed information. Even tick-level data may be fraught with errors. Using summary statistics, such as the mean, standard errors, maximums, minimums, and correlations, tells us a lot about the nature of the data; whether we can really use it, or infer backtest parameter estimates.

When data cleaning is performed, we might ask these questions: what are things to look out for? Are values realistic and logical? How is missing data coded?

Devise a system of reporting data and results. The use of graphs helps the human eye to visualize patterns that might come across as unexpected. Histograms might reveal unexpected distribution, or residual plots might show unexpected prediction error patterns. Scatterplots of residualized data may show additional modeling opportunities.

Residualized data is the difference or "residuals" between the observed values and those of the model.

Data mine your results

From running over several iterations of backtests, the results represent a source of information about your model. Running your model in real-time conditions contains another source of results. By data mining all this wealth of information, we can obtain a data-driven result that can avoid tailoring the model specifications to the sample data. It is recommended that you use shrinkage estimators or model averaging when reporting the results.

Discussion of algorithms in backtesting

After taking into consideration the designing of a backtesting model, one or more algorithms may be used to improve the model on a continuous basis. This section briefly covers some of the algorithmic techniques used in areas of backtesting, such as data mining and machine learning.

K-means clustering

The k-means clustering algorithm is a method of clustering analysis in data mining. From the backtest results of n observations, the k-means algorithm is designed to classify the data into k clusters based on their relative distance from each other. The center point of each cluster is computed. The objective then is to find the within-cluster sum of squares that gives us a model averaged point. The model averaged point indicates the likely average performance of the model, which can be used for further comparison with the performance of other models.

K-nearest neighbor machine learning algorithm

The **k-nearest neighbor** (**KNN**) is a lazy learning technique that does not build any models.

An initial set of backtest model parameters are chosen either by random or best guess.

After analyzing the results of the model, a k number of sets of parameters that is closest to the original set are used for computation in the next step. The model will then take the set of parameters that gives the best results.

The process continues until the terminating condition is reached, thereby always giving the best set of model parameters available.

Classification and regression tree analysis

The **classification and regression tree** (**CART**) analysis contains two decision trees that are used in data mining. The classification tree uses classification rules to classify the outcomes of a model using nodes and branches in the decision tree. The regression tree attempts to assign a real value to the classified outcome. The resulting values are averaged to provide a measure of the quality of the decision.

The 2k factorial design

When designing experiments for backtesting, we can consider the use of 2k factorial design. Suppose we have two factors, A and B. Each factor behaves like a Boolean value, where values of either +1 or -1. A +1 value indicates a quantitatively high value, while -1 indicates a low value. This gives us a combination of $2^2 = 4$ outcomes. For a 3-factor model, this gives us a combination of $2^3 = 8$ outcomes. The following table illustrates an example with two factors with outcomes *W*, *X*, *Y*, and *Z*:

	A	**B**	**Replication I**
Value	+1	+1	W
Value	+1	-1	X
Value	-1	+1	Y
Value	-1	-1	Z

Note that we are generating one replication of backtest to produce a set of outcomes. Performing additional replications gives us more information. From this data, we can perform a regression and analyze its variance. The objectives of these tests are to determine which factors, A or B, are more influential over another, and what values should be chosen so that the outcomes are either near some desired value, able to achieve a low variance, or minimize the effects of uncontrollable variables.

The genetic algorithm

The **genetic algorithm** (**GA**) is a technique where every individual evolves itself through the process of natural selection to optimize a problem. A population of candidate solutions in an optimization problem goes through an iterative process of selection to become parents, undergoing mutation and crossover to produce the next generation of offsprings. Over cycles of successive generations, the population evolves toward an optimal solution.

The application of genetic algorithms can be applied to a variety of optimizing problems, including backtesting, and is especially useful for solving standard optimizations, discontinuous or non-differentiable problems, or nonlinear outcomes.

Summary

A backtest is a simulation of a model-driven investment strategy's response to historical data. The purpose of performing experiments with backtests is to make discoveries about a process or system and to compute various factors related to either risk or return. The factors are typically used together to find a combination that is predictive of return.

While working on designing and developing a backtester, to achieve functionalities, such as simulated market pricing, ordering environment, order matching engine, order book management, as well as account and position updates, we can explore the concept of an event-driven backtesting system.

In this chapter, we designed and implemented an event-driven backtesting system using the `TickData` class, the `MarketDataSource` class, the `Order` class, the `Position` class, the `Strategy` class, the `MeanRevertingStrategy` class, and the `Backtester` class. We plotted our resulting profits and losses onto a graph to help us visualize the performance of our trading strategy.

Backtesting involves a lot of research that merits a literature on its own. In this chapter, we explored ten considerations for designing a backtest model. To help improve our models on a continuous basis, a number of algorithms can be employed in backtesting. We briefly discussed some of these: k-means clustering, k-nearest neighbor, classification and regression tree, 2k factorial design, and genetic algorithm.

In the next chapter, we will discuss Excel with Python, using the Component Object Model (COM).

10
Excel with Python

In finance, Microsoft Excel is used as a handy tool for bond traders and is useful in banking operations, as well as task automations using **Visual Basic for Applications** (**VBA**). Excel supports the use of **Component Object Model** (**COM**) to extend the functionality for custom tasks. This is achieved with the use of COM add-ins as an in-process COM server. With VBA, a wrapper can be created for the COM add-in function so that the COM component can be integrated as a worksheet cell formula function. COM allows the reuse of objects across different software and hardware environments to interface with each other, without the knowledge of its internal implementation. It allows an object to be created in several languages, such as C, C++, Visual Basic, Delphi, or Python.

In this chapter, we will learn how to build a COM server in Python. We will then create a COM client in Microsoft Excel and interface with the COM server to perform numerical pricing on the call and put options. We will use the Black-Scholes model, the binomial tree model, and the trinomial lattice model from the earlier chapters covered in this book for the COM server implementation. By linking to the cell values in Excel, or a market data source subscription within the worksheet cells, we can compute the theoretical option prices on the fly.

In this chapter, we will cover the following topics:

- Overview of the Component Object Model (COM)
- Understanding Excel for finance and COM
- Prerequisites for building a COM server
- Building the Black-Scholes model COM pricing server
- Registering and unregistering the COM server
- Building the Cox-Ross-Rubinstein binomial tree COM server
- Building the trinomial lattice model COM server

- Setting up VBA functions to build a COM client in Excel
- Setting up parameters in Excel to invoke the COM client-server interface
- Computing the theoretical option prices on the fly in Excel

Overview of COM

COM allows the reuse of objects across different software and hardware environments to interface with each other, without the knowledge of its internal implementation. COM is a proprietary standard and is commonly associated with Microsoft's COM. COM forms the basis for Microsoft's other technologies, including ActiveX, COM+, and **Document Component Object Model** (**DCOM**).

COM allows an object to be created in several languages, such as C, C++, Visual Basic, Delphi, or Python. Using COM-aware components, COM classes are built as binary standards. Each COM component has its own class identifier (CLSID), which are globally unique identifiers (GUIDs), used for identification when used on a runtime framework. To locate a COM library, the Microsoft Windows registry is used to list all the available class and interface libraries as GUIDs.

Excel for finance

The spreadsheet application in the Microsoft Office suite was designed for statistical, engineering, and financial data management. In finance, Microsoft Excel is used as a handy tool for bond traders and an integral part of banking operations to task automations using VBA. For example, built-in Excel functions, such as `TBILLYIELD` and `DURATION`, helps you calculate the yield of a T-bill and the Macaulay duration of a bond and displays these values onto a cell.

Excel supports the use of COM to extend the functionality for custom tasks. This is achieved with the use of COM add-ins as an in-process COM server. With VBA, a wrapper can be created for the COM add-in function so that the COM component can be used as a worksheet cell formula function.

In this chapter, we will take a look at building a COM server in Python. We will then use Microsoft Excel, as our source of data parameters, to perform numerical pricing with the COM object. Using this basic example, we can then extend the functionality of the COM objects for many uses, not limited to real-time trading and pricing.

Building a COM server

In this section, we are concerned with building the server component of the COM interface. We will first take a look at the prerequisites for building the server components using Python. Then, we will proceed to build an option pricing the COM server using some of the topics we covered in *Chapter 4*, *Numerical Procedures*.

Prerequisites

The COM interface is an industry standard by Microsoft; therefore, the following software is required to complete this tutorial:

- Microsoft Windows XP operating system or later
- Microsoft Excel 2003 or later
- Python 2.7 or later with SciPy and NumPy packages
- The `pythoncom` module

Getting the pythoncom module

The `pythoncom` module contains Python extensions for Microsoft Windows. The files are available freely as pywin32 on SourceForge at `http://sourceforge.net/projects/pywin32/files/`. To download the executable file, navigate to the **pywin32** folder, and select the latest available build. Download the installer executable that is compatible with your system. Note that there is one download package for each supported version of Python. Be sure to check for the version of Python installed in your environment and download the corresponding package. Some packages have a 32-bit and a 64-bit version available. You must download the one that corresponds to the Python you have installed. Even if you have a 64-bit computer, if you installed a 32-bit version of Python, you must install the 32-bit version of pywin32.

Once the executable file is downloaded to your hard drive, run the installer, and follow the onscreen instructions to add the `pythoncom` module to your Python environment.

Building the Black-Scholes model COM server

Let's build a simple COM server using the classic Black-Scholes options pricing model to calculate the theoretical value of a call or a put option. The calculator is implemented as the `BlackScholes` class with a method named `pricer` that accepts the current underlying price, strike price, annualized interest rate, time left to maturity in terms of years, volatility of the underlying instrument, and annualized dividend yield as its input parameters. The full COM server code in Python is given as follows:

```
""" Black-Scholes pricer COM server """
import numpy as np
import scipy.stats as stats
import pythoncom

class BlackScholes:
    _public_methods_ = ["call_pricer", "put_pricer"]
    _reg_progid_ = "BlackScholes.Pricer"
    _reg_clsid_ =  pythoncom.CreateGuid()

    def d1(self, S0, K, r, T, sigma, div):
        return (np.log(S0/K) + ((r-div) + sigma**2 / 2) * T)/ \
               (sigma * np.sqrt(T))

    def d2(self, S0, K, r, T, sigma, div):
        return (np.log(S0 / K) + ((r-div) - sigma**2 / 2) * T) / \
               (sigma * np.sqrt(T))

    def call_pricer(self, S0, K, r, T, sigma, div):
        d1 = self.d1(S0, K, r, T, sigma, div)
        d2 = self.d2(S0, K, r, T, sigma, div)
        return S0 * np.exp(-div * T) * stats.norm.cdf(d1) \
               - K * np.exp(-r * T) * stats.norm.cdf(d2)

    def put_pricer(self, S0, K, r, T, sigma, div):
        d1 = self.d1(S0, K, r, T, sigma, div)
        d2 = self.d2(S0, K, r, T, sigma, div)
        return K * np.exp(-r * T) * stats.norm.cdf(-d2) \
               - S0 * np.exp(-div * T) *stats.norm.cdf(-d1)

if __name__ == "__main__":
    # Run "python binomial_tree_am.py"
    #   to register the COM server.
```

```
# Run "python binomial_tree_am.py --unregister"
#   to unregister it.
print "Registering COM server..."
import win32com.server.register
win32com.server.register.UseCommandLine(BlackScholes)
```

Note the use of the three magic variables: `_public_methods_`, `_reg_progid_`, and `_reg_clsid_` in the COM server object. The `_public_methods_` variable defines the methods that are exposed to the COM clients. The `_reg_progid_` variable defines the name of the COM server that is called from the COM client. The `_reg_clsid_` variable contains the unique class identifier in the registry.

Registering and unregistering the COM server

Assuming that the code is saved in the `black_scholes.py` file, we can compile the COM server and register with the registry:

```
$ python black_scholes.py
Registering COM server…
Registered: BlackScholes.Pricer
```

The COM server is now accessible for COM communications.

To unregister the COM server, the additional `--unregister` parameter is used:

```
$ python black_scholes.py --unregister
Registering COM server…
Unregistered: BlackScholes.Pricer
```

The COM server is now unregistered and cannot be accessed by the COM clients.

Building the Cox-Ross-Rubinstein binomial tree model COM server

In *Chapter 4*, *Numerical Procedures*, we looked at several options pricing models. One such model is the **Cox-Ross-Rubinstein** (**CRR**) model using a binomial tree. Before we can create a second COM server based on this model, let's copy and paste these class files created earlier, namely, `BinomialCRROption.py`, `BinomialTreeOption.py`, and `StockOption.py`, to our working directory.

Now, let's create our COM server using the `BinomialCRRCOMServer` class and save it as `binomial_crr_com.py`:

```
""" Binomial CRR tree COM server """
from BinomialCRROption import BinomialCRROption
import pythoncom

class BinomialCRRCOMServer:
    _public_methods_ = [ 'pricer']
    _reg_progid_ = "BinomialCRRCOMServer.Pricer"
    _reg_clsid_ = pythoncom.CreateGuid()

    def pricer(self, S0, K, r, T, N, sigma,
               is_call=True, div=0., is_eu=False):
        model = BinomialCRROption(S0, K, r, T, N,
                                  {"sigma": sigma,
                                   "div": div,
                                   "is_call": is_call,
                                   "is_eu": is_eu})
        return model.price()

if __name__ == "__main__":
    print "Registering COM server..."
    import win32com.server.register
    win32com.server.register.UseCommandLine(BinomialCRRCOMServer)
```

Similar to our Black-Scholes COM server, here the `pricer` method creates an instance of the `BinomialCRROption` class and returns the calculated price from the CRR binomial tree model.

Building the trinomial lattice model COM server

In *Chapter 4, Numerical Procedures*, we also explored the use of a trinomial lattice in options pricing. Let's use this model as our third COM server. Let's copy and paste the related class files, namely, `TrinomialLattice.py` and `TrinomialTreeOption.py`, to our working directory.

Create our COM server with the `TrinomialLatticeCOMServer` class and save it as `trinomial_lattice_com.py`:

```
""" Trinomial Lattice COM server """
from TrinomialLattice import TrinomialLattice
import pythoncom
```

```
class TrinomialLatticeCOMServer:
    _public_methods_ = ['pricer']
    _reg_progid_ = "TrinomialLatticeCOMServer.Pricer"
    _reg_clsid_ = pythoncom.CreateGuid()

    def pricer(self, S0, K, r, T, N, sigma,
               is_call=True, div=0., is_eu=False):
        model = TrinomialLattice(S0, K, r, T, N,
                                 {"sigma": sigma,
                                  "div": div,
                                  "is_call": is_call,
                                  "is_eu": is_eu})
        return model.price()

if __name__ == "__main__":
    print "Registering COM server..."
    import win32com.server.register
    win32com.server.register.UseCommandLine(TrinomialLatticeCOMServer)
```

Now, let's build and register our three COM server Python files with the registry:

```
$ python black_scholes.py
Registering COM server…
Registered: BlackScholes.Pricer

$ python binomial_crr_com.py
Registering COM server…
Registered: BinomialCRRCOMServer.Pricer

$ python trinomial_lattice_com.py
Registering COM server…
Registered: TrinomialLatticeCOMServer.Pricer
```

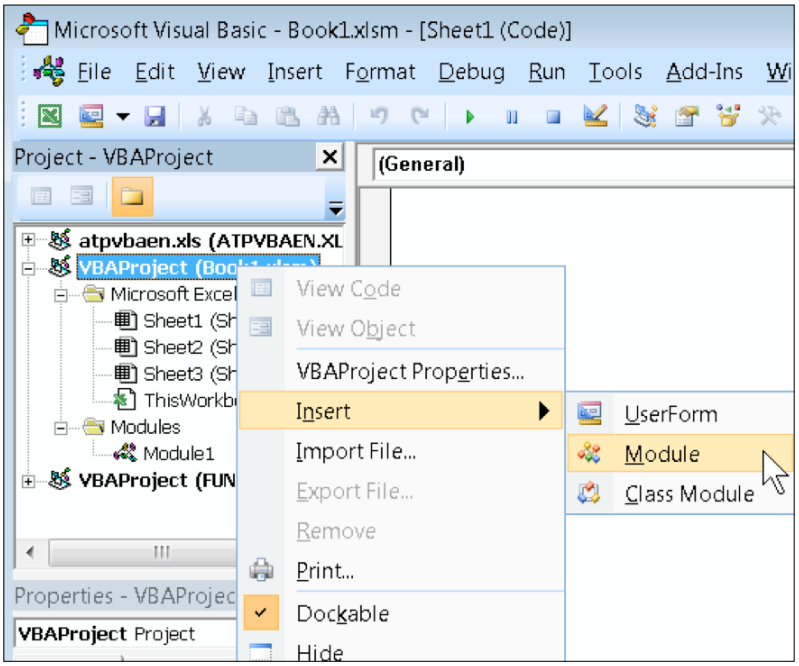
With our COM server components successfully registered with the registry, we can now proceed to create our COM client in Excel in the next section.

Building the COM client in Excel

In the worksheet cells of Microsoft Excel, we can input a number of parameters for a particular option and numerically compute the theoretical option prices using the COM server components we just built in the earlier section. These functions can be made available in the formula cell using Visual Basic. To begin creating these functions, open the *Visual Basic Editor* from Excel by pressing the *Alt + F11* keys on your keyboard.

Setting up the VBA code

In the **Project-VBAProject** toolbar window, right-click on **VBAProject**, select **Insert**, and click on **Module** to insert a new module in the Excel workbook:



In the code editor area, paste the following VBA code:

```
Function BlackScholesOptionPrice( _
     ByVal S0 As Integer, _
     ByVal K As Integer, _
     ByVal r As Double, _
     ByVal T As Double, _
     ByVal sigma As Double, _
```

```
        ByVal dividend As Double, _
        ByVal isCall As Boolean)
        Set BlackScholes = CreateObject("BlackScholes.Pricer")
        If isCall = True Then
            answer = BlackScholes.call_pricer(S0, K, r, T, sigma, \
dividend)
        Else
            answer = BlackScholes.put_pricer(S0, K, r, T, sigma, \
dividend)
        End If
        BlackScholesOptionPrice = answer
 End Function
```

This will create the COM client component of the Black-Scholes model. The `BlackScholesOptionPrice` VBA function takes in a number of input parameters from Excel, which we will define later. The `CreateObject` function is then called and takes the `BlackScholes.Pricer` input string, which is effectively the name, as defined in the `_reg_progid_` variable of the corresponding COM server component. In the COM server, we exposed two methods, `call_pricer` and `put_pricer`, to compute and return the Black-Scholes call and put option prices respectively. The selection of this option is determined by the `isCall` variable, which is true for a call option and false for a put option.

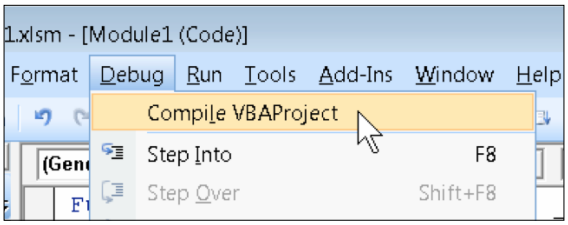
In the same fashion, we can create the COM client functions for our two other pricing methods using the following VBA code:

```
 Function BinomialTreeCRROptionPrice( _
    ByVal S0 As Integer, _
    ByVal K As Integer, _
    ByVal r As Double, _
    ByVal T As Double, _
    ByVal N As Integer, _
    ByVal sigma As Double, _
    ByVal isCall As Boolean, _
    ByVal dividend As Double)
    Set BinCRRTree = CreateObject("BinomialCRRCOMServer.Pricer")
    answer = BinCRRTree.pricer(S0, K, r, T, N, sigma, isCall, _
        dividend, True)
    BinomialTreeCRROptionPrice = answer
End Function
Function TrinomialLatticeOptionPrice( _
    ByVal S0 As Integer, _
    ByVal K As Integer, _
    ByVal r As Double, _
    ByVal T As Double, _
```

```
        ByVal N As Integer, _
        ByVal sigma As Double, _
        ByVal isCall As Boolean, _
        ByVal dividend As Double)
        Set TrinomialLattice = _
            CreateObject("TrinomialLatticeCOMServer.Pricer")
        answer = TrinomialLattice.pricer(S0, K, r, T, N, sigma, _
            isCall, dividend, True)
        TrinomialLatticeOptionPrice = answer
    End Function
```

Here, the `BinomialTreeCRROptionPrice` and `TrinomialLatticeOptionPrice` VBA functions are defined. Similar to the `BlackScholesOptionPrice` function, the `CreateObject` function takes in the string value of `BinomialCRRCOMServer.Pricer` and `TrinomialLatticeCOMServer.Pricer`, as defined in the `_reg_progid_` variable in its respective COM server.

We can compile the code by selecting **Debug** from the toolbar menu and clicking on **Compile VBAProject**:



When the code has been successfully compiled, close the Visual Basic Editor window and return to Excel to input our parameters.

Setting up the cells

Let's assume that we would like to price an option with a strike price of 50. The current underlying price is 50 with a volatility of 0.5 and does not pay dividends. The risk-free rate is 0.05 and the time to maturity is 6 months. We will start with a two-step binomial tree and trinomial lattice with N=2.

In Excel, set up the following cells and values:

	A	B
1	**Parameter**	**Value**
2	S0	50
3	K	50
4	R	0.05
5	T	0.5
6	N	2
7	sigma	0.5
8	Dividend	0.00

We are now ready to price our option using dynamic numerical pricing with COM.

In a new row, set up the following cells and values:

	A	B
10		**Call**
11	**Is call option?**	TRUE
12	**Black-Scholes**	=BlackScholesOptionPrice(B2,B3,B4,B5,B7,B8,B11)
13	**Binomial Tree CRR**	=BinomialTreeCRROptionPrice(B2,B3,B4,B5,B6,B7,B11,B8)
14	**Trinomial Lattice**	=TrinomialLatticeOptionPrice(B2,B3,B4,B5,B6,B7,B11,B8)

Notice that in cells B12 to B14, we are calling the functions that we have defined in the VBA editor. The input values are derived from the values of cells B2 to B8. The Boolean value in B11 determines whether we are pricing a call option or a put option when calling the COM server. Since we are pricing the call options in column B, let's add another column, C to price the put options:

	A	C
10		**Put Price**
11	**Is call option?**	FALSE
12	**Black-Scholes**	=BlackScholesOptionPrice(B2,B3,B4,B5,B7,B8,C11)
13	**Binomial Tree CRR**	=BinomialTreeCRROptionPrice(B2,B3,B4,B5,B6,B7,C11,B8)
14	**Trinomial Lattice**	=TrinomialLatticeOptionPrice(B2,B3,B4,B5,B6,B7,C11,B8)

The formulas are the same, as in the previous table, with the exception of the `isCall` cell reference to C11 instead of B11. This allows us to price a put option.

Our Excel spreadsheet should look something like this:

	A	B	C
1	**Parameters**	**Value**	
2	**S0**	50.00	
3	**K**	50.00	
4	**r**	0.05	
5	**T**	0.5	
6	**N**	2	
7	**sigma**	0.5	
8	**Dividend**	0.00	
9			
10		**Call**	**Put**
11	Is call option?	TRUE	FALSE
12	**Black-Scholes**	7.5636	6.3291
13	**Binomial Tree CRR**	6.7734	5.8685
14	**Trinomial Lattice**	7.1468	6.0823

The call option prices, as computed by the Black-Scholes model, the binomial tree with CRR parameters, and the trinomial lattice model, are **7.5636**, **6.7734**, and **7.1468** respectively. Likewise, the put option prices are **6.3291**, **5.8685**, and **6.0823** respectively.

What happens when we change the value of *N* to a bigger value?

	A	B	C
1	Parameters	Value	
2	S0	50.00	
3	K	50.00	
4	r	0.05	
5	T	0.5	
6	N	1000	
7	sigma	0.5	
8	Dividend	0.00	
9			
10		Call	Put
11	Is call option?	TRUE	FALSE
12	Black-Scholes	7.5636	6.3291
13	Binomial Tree CRR	7.5619	6.4408
14	Trinomial Lattice	7.5627	6.4412

We can see that the values of the binomial tree by the CRR model and the trinomial lattice model converge to the values by the Black-Scholes model as the number of tree step increases.

What else can I do with COM?

On changing the values of `N`, we can see that the values from our custom-defined functions changes on the fly. This makes it possible for dynamic computations of securities, or even real-time numerical pricing, when connected to a market data feed, where values such as `S0` or `K` are changing every second.

The COM server components are separated from each other. Using Python, we can change the implementation of the COM server using the Python modules, such as NumPy or SciPy, to achieve certain aspects of numerical pricing without relying too much on Excel's built-in functions. This also means that we can interchange and interface components that are not related to Excel. The COM model simply provides a transparent bridge between these components and Excel.

Summary

In this chapter, we looked at the use of the Component Object Model (COM) to allow the reuse of objects across different software and hardware environments to interface with each other, without the knowledge of its internal implementation.

To build the server component of the COM interface, we used the `pythoncom` module to create a Black-Scholes pricing COM server with the three magic variables: `_public_methods_`, `_reg_progid_`, and `_reg_clsid_`. Using topics in *Chapter 4*, *Numerical Procedures*, we created COM server components using the binomial tree by the CRR model and trinomial lattice model. We learned how to register and unregister these COM server components with the Windows registry.

In Microsoft Excel, we can input a number of parameters for a particular option and numerically compute the theoretical option prices using the COM server components we built. These functions are made available in the formula cells using Visual Basic. We created the Black-Scholes model, binomial tree CRR model, and trinomial lattice model COM client VBA functions. These functions accept the same input values from the spreadsheet cells to perform numerical pricing on the COM server. We also saw how to update the input parameters in the spreadsheet that dynamically update the option prices on the fly.

Index

Symbols

A

B

C

D

E

F

G

H

I

O

P

Q

R

S

T

V

W

Y

Z

Thank you for buying

Mastering Python for Finance

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

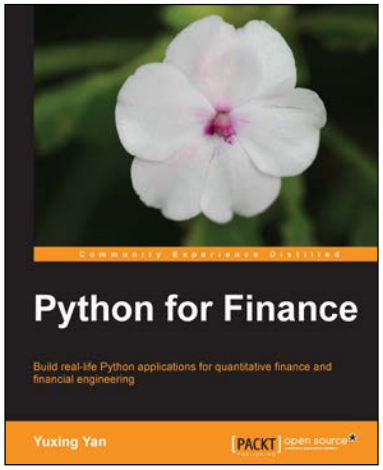
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

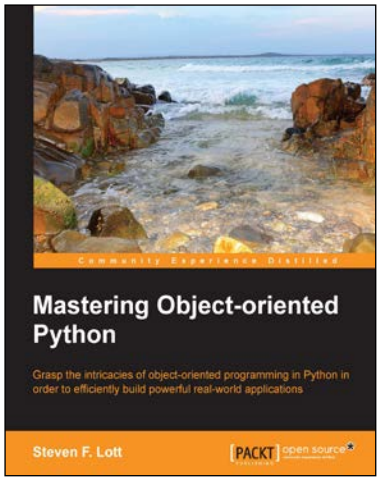


Python for Finance

ISBN: 978-1-78328-437-5 Paperback: 408 pages

Build real-life Python applications for quantitative finance and financial engineering

1. Estimate market risk, form various portfolios, and estimate their variance-covariance matrixes using real-world data.
2. Explains many financial concepts and trading strategies with the help of graphs.
3. A step-by-step tutorial with many Python programs that will help you learn how to apply Python to finance.



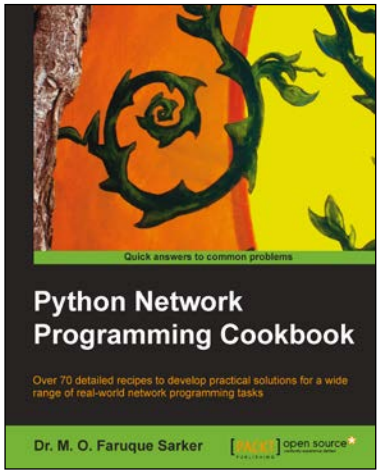
Mastering Object-oriented Python

ISBN: 978-1-78328-097-1 Paperback: 634 pages

Grasp the intricacies of object-oriented programming in Python in order to efficiently build powerful real-world applications

1. Create applications with flexible logging, powerful configuration and command-line options, automated unit tests, and good documentation.
2. Get to grips with different design patterns for the __init__() method.
3. Design callable objects and context managers.
4. Map Python objects to a SQL database using the built-in SQLite module.

[PACKT] open source
community experience distilled
PUBLISHING

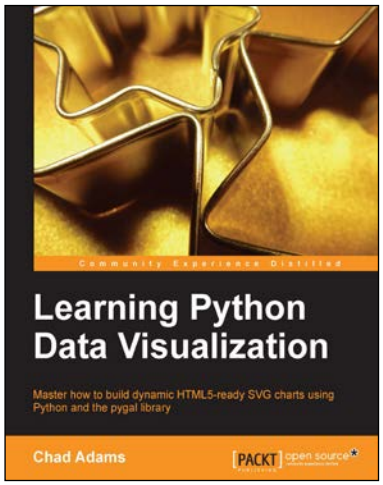


Python Network Programming Cookbook

ISBN: 978-1-84951-346-3 Paperback: 234 pages

Over 70 detailed recipes to develop practical solutions for a wide range of real-world network programming tasks

1. Demonstrates how to write various besopke client/server networking applications using standard and popular third-party Python libraries.
2. Learn how to develop client programs for networking protocols such as HTTP/HTTPS, SMTP, POP3, FTP, CGI, XML-RPC, SOAP and REST.



Learning Python Data Visualization

ISBN: 978-1-78355-333-4 Paperback: 212 pages

Master how to build dynamic HTML5-ready SVG charts using Python and the pygal library

1. A practical guide that helps you break into the world of data visualization with Python.
2. Understand the fundamentals of building charts in Python.
3. Packed with easy-to-understand tutorials for developers who are new to Python or charting in Python.