A STEP-BY-STEP GUIDE TO QUANTITATIVE STRATEGIES

SUCCESSFUL ALGORITHMIC TRADING

Applying the scientific method for profitable trading results.

By Michael L. Halls-Moore

Contents

Limit of Liability/Disclaimer of Warranty

Part I

Introducing Algorithmic Trading

Chapter 1

Introduction to the Book

1.1 Introduction to QuantStart

QuantStart was founded by Michael Halls-Moore, in 2010, to help junior quantitative analysts (QAs) find jobs in the tough economic climate. Since then the site has evolved to become a substantial resource for quantitative finance. The site now concentrates on algorithmic trading, but also discusses quantitative development, in both Python and C++.

Since March 2010, QuantStart has helped over 200,000 visitors improve their quantitative finance skills. You can always contact QuantStart by sending an email to mike@quantstart.com.

1.2 What is this Book?

Successful Algorithmic Trading has been written to teach retail discretionary traders and trading professionals, with basic programming skills, how to create fully automated *profitable* and *robust* algorithmic trading systems using the Python programming language. The book describes the nature of an algorithmic trading system, how to obtain and organise financial data, the concept of backtesting and how to implement an execution system. The book is designed to be extremely practical, with liberal examples of Python code throughout the book to demonstrate the principles and practice of algorithmic trading.

1.3 Who is this Book For?

This book has been written for both retail traders and professional quants who have some basic exposure to programming and wish to learn how to apply modern languages and libraries to algorithmic trading. It is designed for those who enjoy self-study and can learn by example. The book is aimed at individuals interested in actual programming and implementation, as I believe that real success in algorithmic trading comes from fully understanding the implementation details.

Professional quantitative traders will also find the content useful. Exposure to new libraries and implementation methods may lead to more optimal execution or more accurate backtesting.

1.4 What are the Prerequisites?

The book is relatively self-contained, but does assume a familiarity with the basics of trading in a discretionary setting. The book does not require an extensive programming background, but basic familiarity with a programming language is assumed. You should be aware of elementary programming concepts such as variable declaration, flow-control (if-else) and looping (for/while).

Some of the trading strategies make use of statistical machine learning techniques. In addition, the portfolio/strategy optimisation sections make extensive use of search and optimisation algorithms. While a deep understanding of mathematics is not absolutely necessary, it will make it easy to understand how these algorithms work on a conceptual level.

If you are rusty on this material, or it is new to you, have a look at the QuantStart reading list.

1.5 Software/Hardware Requirements

Quantitative trading applications in Python can be developed in Windows, Mac OSX or Linux. This book is agnostic to the operating system so it is best to use whatever system you are comfortable with. I do however recommend Mac OSX or Linux (I use Ubuntu), as I have found installation and data management to be far more straightforward.

In order to write Python programs you simply need access to a text editor (preferably with syntax highlighting). On Windows I tend to use Notepad++. On Mac OSX I make use of SublimeText. On Ubuntu I tend to use emacs, but of course, you can use vim.

The code in this book will run under both Python version 2.7.x (specifically 2.7.6 on my Ubuntu 14.04 machine) and Python 3.4.x (specifically 3.4.0 on my Ubuntu 14.04 machine).

In terms of hardware, you will probably want at least 1GB RAM, but more is always better. You'll also want to use a relatively new CPU and plenty of hard disk storage for historical data, depending upon the frequency you intend to trade at. A 200Gb hard disk should be sufficient for smaller data, while 1TB is useful for a wide symbol universe of tick data.

1.6 Book Structure

The book is designed to create a set of algorithmic trading strategies from idea to automated execution. The process followed is outlined below.

- **Why Algorithmic Trading?** - The benefits of using a systematic/algorithmic approach to trading are discussed as opposed to a discretionary methodology. In addition the different approaches taken to algorithmic trading are shown.
- **Trading System Development** - The process of developing an algorithmic trading system is covered, from hypothesis through to live trading and continual assessment.
- **Trading System Design** - The actual components forming an algorithmic trading system are covered. In particular, signal generation, risk management, performance measurement, position sizing/leverage, portfolio optimisation and execution.
- **Trading System Environment** - The installation procedure of all Python software is carried out and historical data is obtained, cleaned and stored in a local database system.
- **Time Series Analysis** - Various time series methods are used for forecasting, mean-reversion, momentum and volatility identification. These statistical methods later form the basis of trading strategies.
- **Optimisation** - Optimisation/search algorithms are discussed and examples of how they apply to strategy optimisation are considered.
- **Performance Measurement** - Implementations for various measures of risk/reward and other performance metrics are described in detail.
- **Risk Management** - Various sources of risk affecting an algorithmic trading system are outlined and methods for mitigating this risk are provided.
- **Trading Strategy Implementation** - Examples of trading strategies based off statistical measures and technical indicators are provided, along with details of how to optimise a portfolio of such strategies.
- **Execution** - Connecting to a brokerage, creating an automated event-based trading infrastructure and monitoring/resilience tools are all discussed.

1.7 What the Book does not Cover

This is not a beginner book on discretionary trading, nor a book filled with "technical analysis" trading strategies. If you have not carried out any trading (discretionary or otherwise), I would suggest reading some of the books on the QuantStart reading list.

It is also not a Python tutorial book, although once again the QuantStart reading list can be consulted. While every effort has been made to introduce the Python code as each example warrants it, a certain familiarity with Python will be extremely helpful.

1.8 Where to Get Help

The best place to look for help is the articles list on QuantStart.com, found at QuantStart.com/articles or by contacting me at mike@quantstart.com. I've written over 140 articles about quantitative finance (and algorithmic trading in particular), so you can brush up by reading some of these.

I also want to say thank you for purchasing the book and helping to support me while I write more content - it is very much appreciated. Good luck with your algorithmic strategies! Now onto some trading...

Chapter 2

What Is Algorithmic Trading?

Algorithmic trading, as defined here, is the use of an automated system for carrying out trades, which are executed in a pre-determined manner via an algorithm specifically *without any human intervention*. The latter emphasis is important. Algorithmic strategies are designed prior to the commencement of trading and are executed without *discretionary* input from human traders.

In this book "algorithmic trading" refers to the retail practice of automated, systematic and quantitative trading, which will all be treated as synonyms for the purpose of this text. In the financial industry "algorithmic trading" generally refers to a class of execution algorithms (such as Volume Weighted Average Price, VWAP) used to optimise the costs of larger trading orders.

2.1 Overview

Algorithmic trading differs substantially from discretionary trading. In this section the benefits and drawbacks of a *systematic* approach will be outlined.

2.1.1 Advantages

Algorithmic trading possesses numerous advantages over discretionary methods.

Historical Assessment

The most important advantage in creating an automated strategy is that its performance can be ascertained on historical market data, which is (hopefully) representative of future market data. This process is known as **backtesting** and will be discussed in significant depth within this book. Backtesting allows the (prior) statistical properties of the strategy to be determined, providing insight into whether a strategy is likely to be profitable in the future.

Efficiency

Algorithmic trading is substantially more efficient than a discretionary approach. With a fully automated system there is no need for an individual or team to be constantly monitoring the markets for price action or news input. This frees up time for the developer(s) of the trading strategy to carry out more research and thus, depending upon capital constraints, deploy more strategies into a portfolio.

Furthermore by automating the risk management and position sizing process, by considering a stable of systematic strategies, it is necessary to automatically adjust leverage and risk factors dynamically, directly responding to market dynamics in real-time. This is not possible in a discretionary world, as a trader is unable to continuously compute risk and must take occasional breaks from monitoring the market.

No Discretionary Input

One of the primary advantages of an automated trading system is that there is (theoretically) no subsequent discretionary input. This refers to modification of trades at the point of execution or while in a position. Fear and greed can be overwhelming motivators when carrying out discretionary trading. In the context of systematic trading it is rare that discretionary input improves the performance of a strategy.

That being said, it is certainly possible for systematic strategies to stop being profitable due to regime shifts or other external factors. In this instance judgement is required to modify parameters of the strategy or to retire it. Note that this process is still devoid of interfering with individual trades.

Comparison

Systematic strategies provide statistical information on both historical and current performance. In particular it is possible to determine equity growth, risk (in various forms), trading frequency and a myriad of other metrics. This allows an "apples to apples" comparison between various strategies such that capital can be allocated optimally. This is in contrast to the case where only profit & loss (P&L) information is tracked in a discretionary setting, since it masks potential drawdown risk.

Higher Frequencies

This is a corollary of the efficiency advantage discussed above. Strategies that operate at higher frequencies over many markets become possible in an automated setting. Indeed, some of the most profitable trading strategies operate at the ultra-high frequency domain on limit order book data. These strategies are simply impossible for a human to carry out.

2.1.2 Disadvantages

While the advantages of algorithmic trading are numerous there are some disadvantages.

Capital Requirements

Algorithmic trading generally requires a far larger capital base than would be utilised for retail discretionary trading, this is simply due to the fact that there are few brokers who support automated trade execution that do not also require large account minimums. The most prolific brokerage in the retail automated space is Interactive Brokers, who require an account balance of $10,000. The situation is slowly changing, especially as other brokerages are allowing direct connection via the FIX protocol. Further, the Pattern Day Trader requirements as defined by the Securities and Exchange Commission require a minimum of $25,000 in account equity to be maintained at all times, in certain margin situations. These issues will be discussed at length in the section on Execution.

In addition, obtaining data feeds for intraday quantitative strategies, particularly if using futures contracts, is not cheap for the retail trader. Common retail intraday feeds are often priced in the $300-$500 per month range, with commercial feeds an order of magnitude beyond that. Depending upon your latency needs it may be necessary to co-locate a server in an exchange, which increases the monthly costs. For the interday retail trader this is not necessarily an issue, but it is worth considering. There are also ancillaries such as a more robust internet connection and powerful (and thus expensive) desktop machines to be purchased.

Programming/Scientific Expertise

While certain systematic trading platforms exist, such as Quantopian, QuantConnect and TradeStation, that alleviate the majority of the programming difficulty, some do not yet (as of the time of writing) support live execution. TradeStation is clearly an exception in this case. Thus it is a requirement for the algorithmic trader to be relatively proficient both in programming and scientific modelling.

I have attempted to demonstrate a wide variety of strategies, the basis of which are nearly always grounded in a manner that is straightforward to understand. However, if you do possess numerical modelling skills then you will likely find it easier to make use of the statistical time series methods present in the Modelling section. The majority of the techniques demonstrated have already been implemented in external Python libraries, which saves us a substantial amount of development work. Thus we are “reduced” to tying together our data analysis and execution libraries to produce an algorithmic trading system.

2.2 Scientific Method

The design of trading strategies within this book is based solely on the principles of the *scientific method*. The process of the scientific method begins with the formulation of a **question**, based on observations. In the context of trading an example would be "Is there a relationship between the SPDR Gold Shares ETF (GLD) and the Market Vectors Gold Miners ETF (GDX)?". This allows a **hypothesis** to be formed that may explain the observed behaviour. In this instance a hypothesis may be "Does the spread between GLD and GDX have mean-reverting behaviour?". The *null hypothesis* is that there is no mean-reverting behaviour, i.e. the price spread is a *random walk*.

After formulation of a hypothesis it is up to the scientist to disprove the null hypothesis and demonstrate that there is indeed mean reverting behaviour. To carry this out a **prediction** must be defined. Returning to the GLD-GDX example the prediction is that the time series representing the spread of the two ETFs is *stationary*. In order to prove or disprove the hypothesis the prediction is subject to **testing**. In the case of GLD-GDX this means applying statistical stationarity tests such as the Augmented Dickey-Fuller, Hurst Exponent and Variance-Ratio Tests (described in detail in subsequent chapters).

The results of the testing procedure will provide a *statistical* answer upon whether the null hypothesis can be rejected at a certain level of confidence. If the null hypothesis is unable to be rejected, which implies that there was no discernible relationship between the two ETFs, it is still possible that the hypothesis is (partially) true. A larger set of data, incorporation of additional information (such as a third ETF affecting the price) can also be tested. This is the process of **analysis**. It often leads to rejection of the null hypothesis, after refinement.

The primary advantage of using the scientific method for trading strategy design is that if the strategy "breaks down" after a prior period of profitability, it is possible to revisit the initial hypothesis and re-evaluate it, potentially leading to a new hypothesis that leads to regained profitability for a strategy.

This is in direct contrast to the *data mining* or *black box* approach where a large quantity of parameters or "indicators" are applied to a time series. If such a "strategy" is initially profitable and then performance deteriorates it is difficult (if not impossible) to determine why. It often leads to arbitrary application of new information, indicators or parameters that may temporarily lead to profitability but subsequently lead to further performance degradation. In this instance the strategy is usually discarded and the process of "research" continues again.

In this book all trading strategies will be developed with an observation-hypothesis approach.

2.3 Why Python?

The above sections have outlined the benefits of algorithmic trading and the scientific method. It is now time to turn attention to the language of implementation for our trading systems. For this book I have chosen Python. Python is a high-level language designed for speed of development. It possesses a wide array of libraries for nearly any computational task imaginable. It is also gaining wider adoption in the the asset management and investment bank communities.

Here are the reasons why I have chosen Python as a language for trading system research and implementation:

- **Learning** - Python is extremely easy to learn compared to other languages such as C++. You can be extremely productive in Python after only a few weeks (some say days!) of usage.

- **Libraries** - The main reason to use Python is that it comes with a staggering array of libraries, which significantly reduce time to implementation and the chance of introducing bugs into our code. In particular, we will make use of NumPy (vectorised operations), SciPy (optimisation algorithms), pandas (time series analysis), statsmodel (statistical modelling), scikit-learn (statistical/machine learning), IPython (interactive development) and matplotlib (visualisation).
- **Speed of Development** - Python excels at development speed to the extent that some have commented that it is like writing in "pseudocode". The interactive nature of tools like IPython make strategy research extremely rapid, without sacrificing robustness.
- **Speed of Execution** - While not quite as fast as C++, Python provides scientific computing components which are heavily optimised (via vectorisation). If speed of execution becomes an issue one can utilise Cython and obtain execution speeds similar to C, for a small increase in code complexity.
- **Trade Execution** - Python plugins exist for larger brokers, such as Interactive Brokers (IBypy). In addition Python can easily make use of the FIX protocol where necessary.
- **Cost/License** - Python is free, open source and cross-platform. It will run happily on Windows, Mac OSX or Linux.

While Python is extremely applicable to nearly all forms of algorithmic trading, it cannot compete with C (or lower level languages) in the Ultra-High Frequency Trading (UHFT) realm. However, these types of strategies are well outside the scope of this book!

2.4 Can Retail Traders Still Compete?

It is common, as a beginning algorithmic trader practising at retail level, to **question whether it is still possible to compete** with the large institutional quant funds. In this section it will be argued that due to the nature of the institutional regulatory environment, the organisational structure and a need to maintain investor relations, that funds suffer from certain disadvantages that do not concern retail algorithmic traders.

The capital and regulatory constraints imposed on funds lead to certain predictable behaviours, which are able to be exploited by a retail trader. "Big money" moves the markets, and as such one can dream up many strategies to take advantage of such movements. Some of these strategies will be discussed in later chapters. The comparative advantages enjoyed by the algorithmic trader over many larger funds will now be outlined.

2.4.1 Trading Advantages

There are many ways in which a retail algo trader can compete with a fund on their trading process alone, but there are also some disadvantages:

- **Capacity** - A retail trader has greater freedom to play in smaller markets. They can generate significant returns in these spaces, even while institutional funds can't.
- **Crowding the trade** - Funds suffer from "technology transfer", as staff turnover can be high. Non-Disclosure Agreements and Non-Compete Agreements mitigate the issue, but it still leads to many quant funds "chasing the same trade". Whimsical investor sentiment and the "next hot thing" exacerbate the issue. Retail traders are not constrained to follow the same strategies and so can remain uncorrelated to the larger funds.
- **Market impact** - When playing in highly liquid, non-OTC markets, the low capital base of retail accounts reduces market impact substantially.

- **Leverage** - A retail trader, depending upon their legal setup, is constrained by margin/leverage regulations. Private investment funds do not suffer from the same disadvantage, although they are equally constrained from a risk management perspective.
- **Liquidity** - Having access to a prime brokerage is out of reach of the average retail algo trader. They have to "make do" with a retail brokerage such as Interactive Brokers. Hence there is reduced access to liquidity in certain instruments. Trade order-routing is also less clear and is one way in which strategy performance can diverge from backtests.
- **Client news flow** - Potentially the most important disadvantage for the retail trader is lack of access to client news flow from their prime brokerage or credit-providing institution. Retail traders have to make use of non-traditional sources such as meet-up groups, blogs, forums and open-access financial journals.

2.4.2 Risk Management

Retail algo traders often take a different approach to risk management than the larger quant funds. It is often advantageous to be "small and nimble" in the context of risk.

Crucially, there is no risk management budget imposed on the trader beyond that which they impose themselves, nor is there a compliance or risk management department enforcing oversight. This allows the retail trader to deploy custom or preferred risk modelling methodologies, without the need to follow "industry standards" (an implicit investor requirement).

However, the alternative argument is that this flexibility can lead to retail traders to becoming "sloppy" with risk management. Risk concerns may be built-in to the backtest and execution process, without external consideration given to portfolio risk as a whole. Although "deep thought" might be applied to the alpha model (strategy), risk management might not achieve a similar level of consideration.

2.4.3 Investor Relations

Outside investors are the key difference between retail shops and large funds. This drives all manner of incentives for the larger fund - issues which the retail trader need not concern themselves with:

- **Compensation structure** - In the retail environment the trader is concerned only with absolute return. There are no high-water marks to be met and no capital deployment rules to follow. Retail traders are also able to suffer more volatile equity curves since nobody is watching their performance who might be capable of redeeming capital from their fund.
- **Regulations and reporting** - Beyond taxation there is little in the way of regulatory reporting constraints for the retail trader. Further, there is no need to provide monthly performance reports or "dress up" a portfolio prior to a client newsletter being sent. This is a big time-saver.
- **Benchmark comparison** - Funds are not only compared with their peers, but also "industry benchmarks". For a long-only US equities fund, investors will want to see returns in excess of the S&P500, for example. Retail traders are not enforced in the same way to compare their strategies to a benchmark.
- **Performance fees** - The downside to running your own portfolio as a retail trader are the lack of management and performance fees enjoyed by the successful quant funds. There is no "2 and 20" to be had at the retail level!

2.4.4 Technology

One area where the retail trader is at a significant advantage is in the choice of technology stack for the trading system. Not only can the trader pick the "best tools for the job" as they see fit, but there are no concerns about legacy systems integration or firm-wide IT policies. Newer languages such as Python or R now possess packages to construct an end-to-end backtesting, execution, risk and portfolio management system with far fewer lines-of-code (LOC) than may be needed in a more verbose language such as C++.

However, this flexibility comes at a price. One either has to build the stack themselves or outsource all or part of it to vendors. This is expensive in terms of time, capital or both. Further, a trader must debug all aspects of the trading system - a long and potentially painstaking process. All desktop research machines and any co-located servers must be paid for directly out of trading profits as there are no management fees to cover expenses.

In conclusion, it can be seen that retail traders possess significant comparative advantages over the larger quant funds. Potentially, there are many ways in which these advantages can be exploited. Later chapters will discuss some strategies that make use of these differences.

Part II

Trading Systems

Chapter 3

Successful Backtesting

Algorithmic backtesting requires knowledge of many areas, including psychology, mathematics, statistics, software development and market/exchange microstructure. I couldn't hope to cover all of those topics in one chapter, so I'm going to split them into two or three smaller pieces. What will we discuss in this section? I'll begin by defining backtesting and then I will describe the basics of how it is carried out. Then I will elucidate upon the biases we touched upon in previous chapters.

In subsequent chapters we will look at the details of strategy implementations that are often barely mentioned or ignored elsewhere. We will also consider how to make the backtesting process more realistic by including the idiosyncrasies of a *trading exchange*. Then we will discuss transaction costs and how to correctly model them in a backtest setting. We will end with a discussion on the *performance* of our backtests and finally provide detailed examples of common quant strategies.

Let's begin by discussing what backtesting is and why we should carry it out in our algorithmic trading.

3.1 Why Backtest Strategies?

Algorithmic trading stands apart from other types of investment classes because we can more reliably provide expectations about future performance from past performance, as a consequence of abundant data availability. The process by which this is carried out is known as **backtesting**.

In simple terms, backtesting is carried out by exposing your particular strategy algorithm to a stream of historical financial data, which leads to a set of **trading signals**. Each *trade* (which we will mean here to be a 'round-trip' of two signals) will have an associated profit or loss. The accumulation of this profit/loss over the duration of your strategy backtest will lead to the total profit and loss (also known as the 'P & L' or 'PnL'). That is the essence of the idea, although of course the "devil is always in the details"!

What are key reasons for backtesting an algorithmic strategy?

- **Filtration** - If you recall from the previous chapter on Strategy Identification, our goal at the initial research stage was to set up a strategy pipeline and then filter out any strategy that did not meet certain criteria. Backtesting provides us with another filtration mechanism, as we can eliminate strategies that do not meet our performance needs.

- **Modelling** - Backtesting allows us to (safely!) test new models of certain market phenomena, such as transaction costs, order routing, latency, liquidity or other *market microstructure* issues.

- **Optimisation** - Although *strategy optimisation* is fraught with biases, backtesting allows us to increase the performance of a strategy by modifying the quantity or values of the parameters associated with that strategy and recalculating its performance.

- **Verification** - Our strategies are often sourced externally, via our *strategy pipeline*. Backtesting a strategy ensures that it has not been incorrectly implemented. Although we will rarely have access to the signals generated by external strategies, we will often have access to the performance metrics such as the Sharpe Ratio and Drawdown characteristics. Thus we can compare them with our own implementation.

Backtesting provides a host of advantages for algorithmic trading. However, it is not always possible to straightforwardly backtest a strategy. In general, as the frequency of the strategy increases, it becomes harder to correctly model the microstructure effects of the market and exchanges. This leads to less reliable backtests and thus a trickier evaluation of a chosen strategy. This is a particular problem where the execution system is the key to the strategy performance, as with ultra-high frequency algorithms.

Unfortunately, backtesting is fraught with biases of all types and we will now discuss them in depth.

3.2 Backtesting Biases

There are many biases that can affect the performance of a backtested strategy. Unfortunately, these biases have a tendency to inflate the performance rather than detract from it. Thus you should always consider a backtest to be an idealised upper bound on the actual performance of the strategy. It is almost impossible to eliminate biases from algorithmic trading so it is our job to minimise them as best we can in order to make informed decisions about our algorithmic strategies.

There are four major biases that I wish to discuss: *Optimisation Bias*, *Look-Ahead Bias*, *Survivorship Bias* and *Cognitive Bias*.

3.2.1 Optimisation Bias

This is probably the most insidious of all backtest biases. It involves adjusting or introducing additional trading parameters until the strategy performance on the backtest data set is very attractive. However, once live the performance of the strategy can be markedly different. Another name for this bias is "curve fitting" or "data-snooping bias".

Optimisation bias is hard to eliminate as algorithmic strategies often involve many parameters. "Parameters" in this instance might be the entry/exit criteria, look-back periods, averaging periods (i.e the moving average smoothing parameter) or volatility measurement frequency. Optimisation bias can be minimised by keeping the number of parameters to a minimum and increasing the quantity of data points in the training set. In fact, one must also be careful of the latter as older training points can be subject to a prior *regime* (such as a regulatory environment) and thus may not be relevant to your current strategy.

One method to help mitigate this bias is to perform a *sensitivity analysis*. This means varying the parameters incrementally and plotting a "surface" of performance. Sound, fundamental reasoning for parameter choices should, with all other factors considered, lead to a smoother parameter surface. If you have a very jumpy performance surface, it often means that a parameter is not reflecting a phenomena and is an artefact of the test data. There is a vast literature on multi-dimensional optimisation algorithms and it is a highly active area of research. I won't dwell on it here, but keep it in the back of your mind when you find a strategy with a fantastic backtest!

3.2.2 Look-Ahead Bias

Look-ahead bias is introduced into a backtesting system when future data is accidentally included at a point in the simulation where that data would not have actually been available. If we are running the backtest chronologically and we reach time point N, then look-ahead bias occurs if data is included for any point $N + k$, where $k > 0$. Look-ahead bias errors can be incredibly subtle. Here are three examples of how look-ahead bias can be introduced:

- **Technical Bugs** - Arrays/vectors in code often have iterators or index variables. Incorrect *offsets* of these indices can lead to a look-ahead bias by incorporating data at $N + k$ for non-zero k.
- **Parameter Calculation** - Another common example of look-ahead bias occurs when calculating optimal strategy parameters, such as with linear regressions between two time series. If the whole data set (including future data) is used to calculate the regression coefficients, and thus retroactively applied to a trading strategy for optimisation purposes, then future data is being incorporated and a look-ahead bias exists.
- **Maxima/Minima** - Certain trading strategies make use of extreme values in any time period, such as incorporating the high or low prices in OHLC data. However, since these maximal/minimal values can only be calculated at the end of a time period, a look-ahead bias is introduced if these values are used -during- the current period. It is always necessary to lag high/low values by at least one period in any trading strategy making use of them.

As with optimisation bias, one must be extremely careful to avoid its introduction. It is often the main reason why trading strategies underperform their backtests significantly in "live trading".

3.2.3 Survivorship Bias

Survivorship bias is a particularly dangerous phenomenon and can lead to significantly inflated performance for certain strategy types. It occurs when strategies are tested on datasets that do not include the full universe of prior assets that may have been chosen at a particular point in time, but only consider those that have "survived" to the current time.

As an example, consider testing a strategy on a random selection of equities before and after the 2001 market crash. Some technology stocks went bankrupt, while others managed to stay afloat and even prospered. If we had restricted this strategy only to stocks which made it through the market drawdown period, we would be introducing a survivorship bias because they have already demonstrated their success to us. In fact, this is just another specific case of look-ahead bias, as future information is being incorporated into past analysis.

There are two main ways to mitigate survivorship bias in your strategy backtests:

- **Survivorship Bias Free Datasets** - In the case of equity data it is possible to purchase datasets that include delisted entities, although they are not cheap and only tend to be utilised by institutional firms. In particular, Yahoo Finance data is NOT survivorship bias free, and this is commonly used by many retail algo traders. One can also trade on asset classes that are not prone to survivorship bias, such as certain commodities (and their future derivatives).
- **Use More Recent Data** - In the case of equities, utilising a more recent data set mitigates the possibility that the stock selection chosen is weighted to "survivors", simply as there is less likelihood of overall stock delisting in shorter time periods. One can also start building a personal survivorship-bias free dataset by collecting data from current point onward. After 3-4 years, you will have a solid survivorship-bias free set of equities data with which to backtest further strategies.

We will now consider certain psychological phenomena that can influence your trading performance.

3.2.4 Cognitive Bias

This particular phenomena is not often discussed in the context of *quantitative* trading. However, it is discussed extensively in regard to more discretionary trading methods. When creating backtests over a period of 5 years or more, it is easy to look at an upwardly trending equity curve, calculate the compounded annual return, Sharpe ratio and even drawdown characteristics and be satisfied with the results. As an example, the strategy might possess a maximum relative drawdown of 25% and a maximum drawdown duration of 4 months. This would not be atypical for a momentum strategy. It is straightforward to convince oneself that it is easy to tolerate such periods of losses because the overall picture is rosy. However, in practice, it is far harder!

If historical drawdowns of 25% or more occur in the backtests, then in all likelihood you will see periods of similar drawdown in live trading. These periods of drawdown are psychologically difficult to endure. I have observed first hand what an extended drawdown can be like, in an institutional setting, and it is not pleasant - even if the backtests suggest such periods will occur. The reason I have termed it a "bias" is that often a strategy which would otherwise be successful is stopped from trading during times of extended drawdown and thus will lead to significant underperformance compared to a backtest. Thus, even though the strategy is algorithmic in nature, psychological factors can still have a heavy influence on profitability. The takeaway is to ensure that if you see drawdowns of a certain percentage and duration in the backtests, then you should expect them to occur in live trading environments, and will need to persevere in order to reach profitability once more.

3.3 Exchange Issues

3.3.1 Order Types

One choice that an algorithmic trader must make is how and when to make use of the different *exchange orders* available. This choice usually falls into the realm of the *execution system*, but we will consider it here as it can greatly affect strategy backtest performance. There are two types of order that can be carried out: **market orders** and **limit orders**.

A market order executes a trade immediately, irrespective of available prices. Thus large trades executed as market orders will often get a mixture of prices as each subsequent limit order on the opposing side is filled. Market orders are considered *aggressive* orders since they will almost certainly be filled, albeit with a potentially unknown cost.

Limit orders provide a mechanism for the strategy to determine the worst price at which the trade will get executed, with the caveat that the trade may not get filled partially or fully. Limit orders are considered *passive* orders since they are often unfilled, but when they are a price is guaranteed. An individual exchange's collection of limit orders is known as the **limit order book**, which is essentially a queue of buy and sell orders at certain sizes and prices.

When backtesting, it is essential to model the effects of using market or limit orders correctly. For high-frequency strategies in particular, backtests can significantly outperform live trading if the effects of market impact and the limit order book are not modelled accurately.

3.3.2 Price Consolidation

There are particular issues related to backtesting strategies when making use of daily data in the form of Open-High-Low-Close (OHLC) figures, especially for equities. Note that this is precisely the form of data given out by Yahoo Finance, which is a very common source of data for retail algorithmic traders!

Cheap or free datasets, while suffering from survivorship bias (which we have already discussed above), are also often composite price feeds from multiple exchanges. This means that the extreme points (i.e. the open, close, high and low) of the data are very susceptible to "outlying" values due to small orders at regional exchanges. Further, these values are also sometimes more likely to be tick-errors that have yet to be removed from the dataset.

This means that if your trading strategy makes extensive use of any of the OHLC points specifically, backtest performance can differ from live performance as orders might be routed to different exchanges depending upon your broker and your available access to liquidity. The only way to resolve these problems is to make use of higher frequency data or obtain data directly from an individual exchange itself, rather than a cheaper composite feed.

3.3.3 Forex Trading and ECNs

The backtesting of foreign exchange strategies is somewhat trickier to implement than that of equity strategies. Forex trading occurs across multiple venues and Electronic Communication Networks (ECN). The bid/ask prices achieved on one venue can differ substantially from those on another venue. One must be extremely careful to make use of pricing information from the particular venue you will be trading on in the backtest, as opposed to a consolidated feed from multiple venues, as this will be significantly more indicative of the prices you are likely to achieve going forward.

Another idiosyncrasy of the foreign exchange markets is that brokers themselves are not obligated to share *trade* prices/sizes with every trading participant, since this is their proprietary information[6]. Thus it is more appropriate to use bid-ask quotes in your backtests and to be extremely careful of the variation of transaction costs between brokers/venues.

3.3.4 Shorting Constraints

When carrying out short trades in the backtest it is necessary to be aware that some equities may not have been available (due to the lack of availability in that stock to borrow) or due to a market constraint, such as the US SEC banning the shorting of financial stocks during the 2008 market crisis.

This can severely inflate backtesting returns so be careful to include such short sale constraints within your backtests, or avoid shorting at all if you believe there are likely to be liquidity constraints in the instruments you trade.

3.4 Transaction Costs

One of the most **prevalent beginner mistakes** when implementing trading models is to neglect (or grossly underestimate) the effects of *transaction costs* on a strategy. Though it is often assumed that transaction costs only reflect broker commissions, there are in fact many other ways that costs can be accrued on a trading model. The three main types of costs that must be considered include:

3.4.1 Commission

The most direct form of transaction costs incurred by an algorithmic trading strategy are commissions and fees. All strategies require some form of access to an *exchange*, either directly or through a brokerage intermediary ("the broker"). These services incur an incremental cost with each trade, known as *commission*.

Brokers generally provide many services, although quantitative algorithms only really make use of the exchange infrastructure. Hence brokerage commissions are often small on per trade basis. Brokers also charge *fees*, which are costs incurred to clear and settle trades. Further to this are *taxes* imposed by regional or national governments. For instance, in the UK there is a *stamp duty* to pay on equities transactions. Since commissions, fees and taxes are generally fixed, they are relatively straightforward to implement in a backtest engine (see below).

3.4.2 Slippage

Slippage is the difference in price achieved between the time when a trading system decides to transact and the time when a transaction is actually carried out at an exchange. Slippage is a considerable component of transaction costs and can make the difference between a very profitable strategy and one that performs poorly. Slippage is a function of the underlying asset volatility, the latency between the trading system and the exchange and the type of strategy being carried out.

An instrument with higher volatility is more likely to be moving and so prices between signal and execution can differ substantially. Latency is defined as the time difference between signal generation and point of execution. Higher frequency strategies are more sensitive to latency issues and improvements of milliseconds on this latency can make all the difference towards profitability. The type of strategy is also important. Momentum systems suffer more from slippage on average because they are trying to purchase instruments that are already moving in the forecast direction. The opposite is true for mean-reverting strategies as these strategies are moving in a direction opposing the trade.

3.4.3 Market Impact

Market impact is the cost incurred to traders due to the supply/demand dynamics of the exchange (and asset) through which they are trying to trade. A large order on a relatively illiquid asset is likely to *move the market* substantially as the trade will need to access a large component of the current supply. To counter this, large block trades are broken down into smaller "chunks" which are transacted periodically, as and when new liquidity arrives at the exchange. On the opposite end, for highly liquid instruments such as the S&P500 E-Mini index futures contract, low volume trades are unlikely to adjust the "current price" in any great amount.

More illiquid assets are characterised by a larger **spread**, which is the difference between the current bid and ask prices on the **limit order book**. This spread is an additional transaction cost associated with any trade. Spread is a very important component of the total transaction cost - as evidenced by the myriad of UK spread-betting firms whose advertising campaigns express the "tightness" of their spreads for heavily traded instruments.

3.5 Backtesting vs Reality

In summary there are a staggering array of factors that can be simulated in order to generate a realistic backtest. The dangers of overfitting, poor data cleansing, incorrect handling of transaction costs, market regime change and trading constraints often lead to a backtest performance that differs substantially from a live strategy deployment.

Thus one must be very aware that future performance is very unlikely to match historical performance directly. We will discuss these issues in further detail when we come to implement an event-driven backtesting engine near the end of the book.

Chapter 4

Automated Execution

Automated execution is the process of letting the strategy automatically generate execution signals that are the sent to the broker without any human intervention. This is the purest form of algorithmic trading strategy, as it minimises issues due to human intervention. It is the type of system that we will consider most often during this book.

4.1 Backtesting Platforms

The software landscape for strategy backtesting is vast. Solutions range from fully-integrated institutional grade sophisticated software through to programming languages such as C++, Python and R where nearly everything must be written from scratch (or suitable 'plugins' obtained). As quant traders we are interested in the balance of being able to "own" our trading technology stack versus the speed and reliability of our development methodology. Here are the key considerations for software choice:

- **Programming Skill** - The choice of environment will in a large part come down to your ability to program software. I would argue that being in control of the total stack will have a greater effect on your long term PnL than outsourcing as much as possible to vendor software. This is due to the downside risk of having external bugs or idiosyncrasies that you are unable to fix in vendor software, which would otherwise be easily remedied if you had more control over your "tech stack". You also want an environment that strikes the right balance between productivity, library availability and speed of execution. I make my own personal recommendation below.
- **Execution Capability/Broker Interaction** - Certain backtesting software, such as Tradestation, ties in directly with a brokerage. I am not a fan of this approach as reducing transaction costs are often a big component of getting a higher Sharpe ratio. If you're tied into a particular broker (and Tradestation "forces" you to do this), then you will have a harder time transitioning to new software (or a new broker) if the need arises. Interactive Brokers provide an API which is robust, albeit with a slightly obtuse interface.
- **Customisation** - An environment like MATLAB or Python gives you a great deal of flexibility when creating algo strategies as they provide fantastic libraries for nearly any mathematical operation imaginable, but also allow extensive customisation where necessary.
- **Strategy Complexity** - Certain software just isn't cut out for heavy number crunching or mathematical complexity. Excel is one such piece of software. While it is good for simpler strategies, it cannot really cope with numerous assets or more complicated algorithms, at speed.
- **Bias Minimisation** - Does a particular piece of software or data lend itself more to trading biases? You need to make sure that if you want to create all the functionality yourself,

that you don't introduce bugs which can lead to biases. An example here is look-ahead bias, which Excel minimises, while a vectorised research backtester might lend itself to accidentally.

- **Speed of Development** - One shouldn't have to spend months and months implementing a backtest engine. Prototyping should only take a few weeks. Make sure that your software is not hindering your progress to any great extent, just to grab a few extra percentage points of execution speed.
- **Speed of Execution** - If your strategy is completely dependent upon execution timeliness (as in HFT/UHFT) then a language such as C or C++ will be necessary. However, you will be verging on Linux kernel optimisation and FPGA usage for these domains, which is outside the scope of the book.
- **Cost** - Many of the software environments that you can program algorithmic trading strategies with are completely free and open source. In fact, many hedge funds make use of open source software for their entire algo trading stacks. In addition, Excel and MATLAB are both relatively cheap and there are even free alternatives to each.

Different strategies will require different software packages. HFT and UHFT strategies will be written in C/C++. These days such strategies are often carried out on GPUs and FPGAs. Conversely, low-frequency directional equity strategies are easy to implement in TradeStation, due to the "all in one" nature of the software/brokerage.

4.1.1 Programming

Custom development of a backtesting language within a first-class programming language provides the most flexibility when testing a strategy. Conversely, a vendor-developed integrated backtesting platform will always have to make assumptions about how backtests are carried out. The choice of available programming languages is large and diverse. It is not obvious before development which language would be suitable.

Once a strategy has been codified into systematic rules it is necessary to backtest it in such a manner that the quantitative trader is confident that its future performance will be reflective of its past performance. There are generally two forms of backtesting system that are utilised to test this hypothesis. Broadly, they are categorised as *research back testers* and *event-driven back testers*.

4.1.2 Research Tools

The simpler form of a backtesting tool, the research tool, is usually considered first. The research tool is used to quickly ascertain whether a strategy is likely to have any performance. Such tools often make unrealistic assumptions about transaction costs, likely fill prices, shorting constraints, venue dependence, risk management and a host of other issues that were outlined in the previous chapter. Common tools for research include MATLAB, R, Python and Excel.

The research stage is useful because the software packages provide significant *vectorised* capability, which leads to good execution speed and straightforward implementation (less lines of code). Thus it is possible to test multiple strategies, combinations and variants in a rapid, iterative manner.

While such tools are often used for both backtesting and execution, such research environments are generally not suitable for strategies that approach intraday trading at higher frequencies (sub-minute). This is because these environments do not often possess the necessary libraries to connect to real-time market data vendor servers or can interface with brokerage APIs in a clean fashion.

Despite these executional shortcomings, research environments are heavily used within the professional quantitative environment. They are the "firs test" for all strategy ideas before promoting them to a more rigourous check within a realistic backtesting environment.

4.1.3 Event-Driven Backtesting

Once a strategy has been deemed suitable on a research basis it must be tested in a more realistic fashion. Such realism attempts to account for the majority (if not all) of the issues described in the previous chapter. The ideal situation is to be able to use the same trade generation code for historical backtesting as well as live execution. This can be achieved using an *event-driven backtester*.

Event-driven systems are widely used in software engineering, commonly for handling graphical user interface (GUI) input within window-based operating systems. They are also ideal for algorithmic trading. Such systems are often written in high-performance languages such as C++, C# and Java.

Consider a situation where an automated trading strategy is connected to a real-time market feed and a broker (these two may be one and the same). New market information will be sent to the system, which triggers and *event* to generate a new trading signal and thus an execution *event*. Thus such a system is in a continuous loop waiting to receive events and handle them appropriately.

It is possible to generate sub-components such as a historic data handler and brokerage simulator, which can mimic their live counterparts. This allows backtesting strategies in a manner that is extremely similar to live execution.

The disadvantage of such systems is that they are far more complicated to design and implement than a simpler research tool. Thus the "time to market" is longer. They are more prone to bugs and require a reasonable knowledge of programming and, to a degree, software development methodology.

4.1.4 Latency

In engineering terms, *latency* is defined as the time interval between a simulation and a response. For our purposes it will generally refer to the round-trip time delay between the generation of an execution signal and the receipt of the fill information from a broker that is carrying out the execution.

Such latency is rarely an issue on low-frequency interday strategies since the likely price movement during the latency period will not affect the strategy to any great extent. Unfortunately, the same is not true of higher-frequency strategies. At these frequencies latency becomes important. The ultimate goal is to reduce latency as much as possible in order to minimise *slippage*, as discussed in the previous chapter.

Decreasing latency involves minimising the "distance" between the algorithmic trading system and the ultimate exchange on which an order is being executed. This can involve shortening the geographic distance between systems (and thus reducing travel down network cabling), reducing the processing carried out in networking hardware (important in HFT strategies) or choosing a brokerage with more sophisticated infrastructure.

Decreasing latency becomes exponentially more expensive as a function of "internet distance" (i.e. the network distance between two servers). Thus, for a high-frequency trader, a compromise must be reached between expenditure of latency-reduction vs the gain from minimising slippage. These issues will be discussed in the section on *Colocation* below.

4.1.5 Language Choices

Some issues that drive language choice have already been outlined. Now we will consider the benefits and drawbacks of individual programming languages. I have broadly categorised the languages into high-performance/harder development vs lower-performance/easier development. These are subjective terms and some will disagree depending upon their background.

One of the most important aspects of programming a custom backtesting environment is that the programmer is familiar with the tools being used. That is probably a more important criterion than speed of development. However, for those that are new to the programming language landscape, the following should clarify what tends to be utilised within algorithmic trading.

C++, C# and Java

C++, C# and Java are all examples of *general purpose* object-oriented programming languages. That is, they can be used without a corresponding IDE, are all cross-platform (can be run on Windows, Mac OSX or Linux), have a wide range of libraries for nearly any imaginable task and possess have rapid execution speed.

If ultimate execution speed is desired then C++ (or C) is likely to be the best choice. It offers the most flexibility for managing memory and optimising execution speed. This flexibility comes at a price. C++ is notoriously tricky to learn well and can often lead to subtle bugs. Development time can take much longer than in other languages.

C# and Java are similar in that they both require all components to be objects, with the exception of primitive data types such as floats and integers. They differ from C++ in that they both perform automatic *garbage collection*. This means that memory does not have to be manually de-allocated upon destruction of an object. Garbage collection adds a performance overhead but it makes development substantially more rapid. These languages are both good choices for developing a backtester as they have native GUI capabilities, numerical analysis libraries and fast execution speed.

Personally, I would make use of C++ for creating an event-driven backtester that needs extremely rapid execution speed, such as for HFT. This is only if I feel that a Python event-driven system was becoming a bottleneck, as the latter language would be my first choice for such a system.

MATLAB, R and Python

MATLAB is a commercial integrated development environment (IDE) for numerical computation. It has gained wide acceptance in the academic, engineering and financial sectors. It has a huge array of numerical libraries for many scientific computing tasks and possesses a rapid execution speed, assuming that any algorithm being developed is subject to *vectorisation* or *parallelisation*. It is expensive and this sometimes makes it less appealing to retail traders on a budget. MATLAB is sometimes used for direct execution through to a brokerage such as Interactive Brokers.

R is less of a general purpose programming language and more of a statistics scripting environment. It is free, open-source, cross-platform and contains a huge array of freely-available statistical packages for carrying out extremely advanced analysis. R is very widely used in the quantitative hedge fund industry for statistical/strategy research. While it is possible to connect R to a brokerage, is not well suited to the task and should be considered more of a research tool. In addition, it lacks execution speed unless operations are vectorised.

I've grouped Python under this heading, although it sits somewhere between MATLAB, R and the aforementioned general-purpose languages. It is free, open-source and cross-platform. It is *interpreted* as opposed to *compiled*, which makes it natively slower than C++. Despite this it contains a library for carrying out nearly any task imaginable, from scientific computing through to low-level web server design. In particular, it contains NumPy, SciPy, pandas, matplotlib and scikit-learn, which provide a robust numerical research environment that, when vectorised, is comparable to compiled language execution speed.

Further, it has mature libraries for connecting to brokerages. This makes it a "one-stop shop" for creating an event-driven backtesting and live execution environment without having to step into other languages. Execution speed is more than sufficient for intraday traders trading on the time scale of minutes. Finally, it is very straightforward to pick up and learn, when compared to lower-level languages like C++. For these reasons we make extensive use of Python within this book.

4.1.6 Integrated Development Environments

The term IDE has multiple meanings within algorithmic trading. Software developers use it to mean a GUI that allows programming with syntax highlighting, file browsing, debugging and code execution features. Algorithmic traders use it to mean a fully-integrated backtesting/trading environment, including historical or real-time data download, charting, statistical evaluation and live execution. For our purposes, I use the term to mean any environment (often GUI-based) that is *not* a general purpose programming language, such as C++ or Python. MATLAB is considered an IDE, for instance.

Excel

While some purer quants may look down on Excel, I have found it to be extremely useful for "sanity checking" of results. The fact that all of the data is directly available, rather than hidden behind objects, makes it straightforward to implement very basic signal/filter strategies. Brokerages, such as Interactive Brokers, also allow DDE plugins that allow Excel to receive real-time market data and execute trading orders.

Despite the ease of use, Excel is extremely slow for any reasonable scale of data or level of numerical computation. I only use it to error-check when developing against other strategies and to make sure I've avoided look-ahead bias, which is easy to see in Excel due to the spreadsheet nature of the software.

If you are uncomfortable with programming languages and are carrying out an interday strategy, then Excel may be the perfect choice.

Commercial/Retail Backtesting Software

The market for retail charting, "technical analysis" and backtesting software is extremely competitive. Features offersd by such software include real-time charting of prices, a wealth of technical indicators, customised backtesting langauges and automated execution.

Some vendors provide an all-in-one solution, such as TradeStation. TradeStation are an online brokerage who produce trading software (also known as TradeStation) that provides electronic order execution across multiple asset classes. I don't currently believe that they offer a direct API for automated execution, rather orders must be placed through the software. This is in contrast to Interactive Brokers, who have a more stripped down charting/trading interface (Trader WorkStation), but offer both their proprietary real-time market/order execution APIs and a FIX interface.

Another extremely popular platform is MetaTrader, which is used in foreign exchange trading for creating 'Expert Advisors'. These are custom scripts written in a proprietary language that can be used for automated trading. I haven't had much experience with either TradeStation or MetaTrader so I won't spend too much time discussing their merits.

Such tools are useful if you are not comfortable with in depth software development and wish a lot of the details to be taken care of. However, with such systems a lot of flexibility is sacrificed and you are often tied to a single brokerage.

Web-Based Tools

The two current popular web-based backtesting systems are Quantopian (https://www.quantopian.com/) and QuantConnect (https://www.quantconnect.com/). The former makes use of Python (and ZipLine, see below) while the latter utilises C#. Both provide a wealth of historical data. Quantopian currently supports live trading with Interactive Brokers, while QuantConnect is working towards live trading.

Open Source Backtesting Software

In addition to the commercial offerings, there are open source alternatives for backtesting software.

Algo-Trader is a Swiss-based firm that offer both an open-source and a commercial license for their system. From what I can gather the offering seems quite mature and they have many institutional clients. The system allows full historical backtesting and *complex event processing* and they tie into Interactive Brokers. The Enterprise edition offers substantially more high performance features.

Marketcetera provide a backtesting system that can tie into many other languages, such as Python and R, in order to leverage code that you might have already written. The 'Strategy

Studio' provides the ability to write backtesting code as well as optimised execution algorithms and subsequently transition from a historical backtest to live paper trading.

ZipLine is the Python library that powers the Quantopian service mentioned above. It is a fully event-driven backtest environment and currently supports US equities on a minutely-bar basis. I haven't made extensive use of ZipLine, but I know others who feel it is a good tool. There are still many areas left to improve, but the team are constantly working on the project so it is very actively maintained.

There are also some Github/Google Code hosted projects that you may wish to look into. I have not spent any great deal of time investigating them. Such projects include OpenQuant (http://code.google.com/p/openquant/), TradeLink (https://code.google.com/p/tradelink/) and PyAlgoTrade (http://gbeced.github.io/pyalgotrade/).

Institutional Backtesting Software

Institutional-grade backtesting systems, such as Deltix and QuantHouse, are not often utilised by retail algorithmic traders. The software licenses are generally well outside the budget for infrastructure. That being said, such software is widely used by quant funds, proprietary trading houses, family offices and the like.

The benefits of such systems are clear. They provide an all-in-one solution for data collection, strategy development, historical backtesting and live execution across single instruments or portfolios, up to the ultra-high frequency level. Such platforms have had extensive testing and plenty of "in the field" usage and so are considered robust.

The systems are event-driven and as such the backtesting environment can often simulate the live environment well. The systems also support optimised execution algorithms, which attempt to minimise transaction costs.

I have to admit that I have not had much experience of Deltix or QuantHouse beyond some cursory overviews. That being said, the budget alone puts them out of reach of most retail traders, so I won't dwell on these systems.

4.2 Colocation

The software landscape for algorithmic trading has now been surveyed. It is now time to turn attention towards implementation of the hardware that will execute our strategies.

A retail trader will likely be executing their strategy from home during market hours, turning on their PC, connecting to the brokerage, updating their market software and then allowing the algorithm to execute automatically during the day. Conversely, a professional quant fund with significant *assets under management* (AUM) will have a dedicated exchange-colocated server infrastructure in order to reduce latency as far as possible to execute their high speed strategies.

4.2.1 Home Desktop

The simplest approach to hardware deployment is simply to carry out an algorithmic strategy with a home desktop computer connected to the brokerage via a broadband (or similar) connection.

While this approach is straightforward to get started it does suffers from many drawbacks. Primarily, the desktop machine is subject to power failure, unless backed up by a UPS. In addition, a home internet connection is also at the mercy of the ISP. Power loss or internet connectivity failure could occur at a crucial moment in trading, leaving the algorithmic trader with open positions that are unable to be closed.

Secondly, a desktop machine must occasionally be restarted, often due to the reliability of the operating system. This means that the strategy suffers from a degree of indirect manual intervention. If this occurs outside of trading hours the problem is mitigated. However, if a computer needs a restart during trading hours the problem is similar to a power loss. Unclosed positions may still be subject to risk.

Component failure also leads to the same set of "downtime" problems. A failure in the hard disk, monitor or motherboard often occurs at precisely the wrong time. For all of these reasons

I hesitate to recommend a home desktop approach to algorithmic trading. If you do decide to pursue this approach, make sure to have both a backup computer AND a backup internet connection (e.g. a 3G dongle) that you can use to close out positions under a downtime situation.

4.2.2 VPS

The next level up from a home desktop is to make use of a **virtual private server** (VPS). A VPS is a remote server system often marketed as a "cloud" service. They are far cheaper than a corresponding dedicated server, since a VPS is actually a partition of a much larger server, with a virtual isolated operating system environment solely available to you. CPU load is shared between multiple servers and a portion of the systems RAM is allocated to the VPS.

Common VPS providers include Amazon EC2 and Rackspace Cloud. They provide entry-level systems with low RAM and basic CPU usage, through to enterprise-ready high RAM, high CPU servers. For the majority of algorithmic retail traders, the entry level systems suffice for low-frequency intraday or interday strategies and smaller historical data databases.

The benefits of a VPS-based system include 24/7 availability (with a certain realistic downtime!), more robust monitoring capabilities, easy "plugins" for additional services, like file storage or managed databases and a flexible architecture. The drawbacks include expense as the system grows, since dedicated hardware becomes far cheaper per performance, assuming colocation away from an exchange, as well as handling failure scenarios (i.e. by creating a second identical VPS, for instance).

In addition, latency is not always improved by choosing a VPS/cloud provider. Your home location may be closer to a particular financial exchange than the data centres of your cloud provider. This is somewhat mitigated by choosing a firm that provide VPS geared specifically for algorithmic trading which are located at or near exchanges, however these will likely cost more than a "traditional" VPS provider such as Amazon or Rackspace.

4.2.3 Exchange

In order to get the best latency minimisation and fastest systems, it is necessary to colocate a dedicated server (or set of servers) directly at the exchange data centre. This is a prohibitively expensive option for nearly all retail algorithmic traders (unless they're very well capitalised). It is really the domain of the professional quantitative fund or brokerage.

As I mentioned above, a more realistic option is to purchase a VPS system from a provider that is located near an exchange. I won't dwell too heavily on exchange colocation, as the topic is somewhat outside the scope of the book.

Chapter 5

Sourcing Strategy Ideas

In this chapter I want to introduce you to the methods by which I myself identify profitable algorithmic trading strategies. We will discuss how to find, evaluate and select such systems. I'll explain how identifying strategies is as much about personal preference as it is about strategy performance, how to determine the type and quantity of historical data for testing, how to dispassionately evaluate a trading strategy and finally how to proceed towards the backtesting phase and strategy implementation.

5.1 Identifying Your Own Personal Preferences for Trading

In order to be a successful trader - either discretionally or algorithmically - it is necessary to ask yourself some honest questions. Trading provides you with the ability to lose money at an alarming rate, so it is necessary to "know thyself" as much as it is necessary to understand your chosen strategy.

I would say the most important consideration in trading is **being aware of your own personality**. Trading, and algorithmic trading in particular, requires a significant degree of discipline, patience and emotional detachment. Since you are letting an algorithm perform your trading for you, it is necessary to be resolved not to interfere with the strategy when it is being executed. This can be extremely difficult, especially in periods of extended drawdown. However, many strategies that have been shown to be highly profitable in a backtest can be ruined by simple interference. Understand that if you wish to enter the world of algorithmic trading you will be emotionally tested and that in order to be successful, it is necessary to work through these difficulties!

The next consideration is one of **time**. Do you have a full time job? Do you work part time? Do you work from home or have a long commute each day? These questions will help determine the *frequency* of the strategy that you should seek. For those of you in full time employment, an intraday futures strategy may not be appropriate (at least until it is fully automated!). Your time constraints will also dictate the methodology of the strategy. If your strategy is frequently traded and reliant on expensive news feeds (such as a Bloomberg terminal) you will clearly have to be realistic about your ability to successfully run this while at the office! For those of you with a lot of time, or the skills to automate your strategy, you may wish to look into a more technical high-frequency trading (HFT) strategy.

My belief is that it is necessary to carry out **continual research** into your trading strategies to maintain a consistently profitable portfolio. Few strategies stay "under the radar" forever. Hence a significant portion of the time allocated to trading will be in carrying out ongoing research. Ask yourself whether you are prepared to do this, as it can be the difference between strong profitability or a slow decline towards losses.

You also need to consider your **trading capital**. The generally accepted ideal minimum amount for a quantitative strategy is 50,000 USD (approximately £35,000 for us in the UK). If I was starting again, I would begin with a larger amount, probably nearer 100,000 USD (approximately £70,000). This is because transaction costs can be extremely expensive for mid- to high-frequency strategies and it is necessary to have sufficient capital to absorb them in times of drawdown. If you are considering beginning with less than 10,000 USD then you will need to restrict yourself to low-frequency strategies, trading in one or two assets, as transaction costs will rapidly eat into your returns. *Interactive Brokers, which is one of the friendliest brokers to those with programming skills, due to its API, has a retail account minimum of 10,000 USD.*

Programming skill is an important factor in creating an automated algorithmic trading strategy. Being knowledgeable in a programming language such as C++, Java, C#, Python or R will enable you to create the end-to-end data storage, backtest engine and execution system yourself. This has a number of advantages, chief of which is the ability to be completely aware of all aspects of the trading infrastructure. It also allows you to explore the higher frequency strategies as you will be in full control of your "technology stack". While this means that you can test your own software and eliminate bugs, it also means more time spent coding up infrastructure and less on implementing strategies, at least in the earlier part of your algo trading career. You may find that you are comfortable trading in Excel or MATLAB and can outsource the development of other components. I would not recommend this however, particularly for those trading at high frequency.

You need to ask yourself **what you hope to achieve** by algorithmic trading. Are you interested in a regular income, whereby you hope to draw earnings from your trading account? Or, are you interested in a long-term capital gain and can afford to trade without the need to drawdown funds? Income dependence will dictate the frequency of your strategy. More regular income withdrawals will require a higher frequency trading strategy with less volatility (i.e. a higher Sharpe ratio). Long-term traders can afford a more sedate trading frequency.

Finally, do not be deluded by the notion of becoming extremely wealthy in a short space of time! **Algo trading is NOT a get-rich-quick scheme** - if anything it can be a become-poor-quick scheme. It takes significant discipline, research, diligence and patience to be successful at algorithmic trading. It can take months, if not years, to generate consistent profitability.

5.2 Sourcing Algorithmic Trading Ideas

Despite common perceptions to the contrary, it is actually quite straightforward to locate profitable trading strategies in the public domain. Never have trading ideas been more readily available than they are today. Academic finance journals, pre-print servers, trading blogs, trading forums, weekly trading magazines and specialist texts provide thousands of trading strategies with which to base your ideas upon.

Our goal as *quantitative trading researchers* is to establish a **strategy pipeline** that will provide us with a stream of ongoing trading ideas. Ideally we want to create a methodical approach to sourcing, evaluating and implementing strategies that we come across. The aims of the *pipeline* are to generate a consistent quantity of new ideas and to provide us with a framework for rejecting the majority of these ideas with the minimum of emotional consideration.

We must be extremely careful not to let cognitive biases influence our decision making methodology. This could be as simple as having a preference for one asset class over another (gold and other precious metals come to mind) because they are perceived as more exotic. Our goal should always be to find consistently profitable strategies, with positive expectation. The choice of asset class should be based on other considerations, such as trading capital constraints, brokerage fees and leverage capabilities.

5.2.1 Textbooks

If you are completely unfamiliar with the concept of a *trading strategy* and financial markets in general then the first place to look is with established textbooks. Classic texts provide a wide range of simpler, more straightforward ideas, with which to familiarise yourself with quantitative trading. Here is a selection that I recommend for those who are new to quantitative trading, which gradually become more sophisticated as you work through the list.

Financial Markets and Participants

The following list details books that outline how capital markets work and describe modern electronic trading.

- **Financial Times Guide to the Financial Markets** by Glen Arnold[1] - This book is designed for the novice to the financial markets and is extremely useful for gaining insight into all of the market participants. For our purposes, it provides us with a list of markets on which we might later form algorithmic trading strategies.
- **Trading and Exchanges: Market Microstructure for Practitioners** by Larry Harris[7] - This is an extremely informative book on the participants of financial markets and how they operate. It provides significant detail in how trades are carried out, the various motives of the players and how markets are regulated. While some may consider it "dry reading" I believe it is absolutely essential to understand these concepts to be a good algorithmic trader.
- **Algorithmic Trading and DMA: An introduction to direct access trading strategies** by Barry Johnson[10] - Johnson's book is geared more towards the technological side of markets. It discusses order types, optimal execution algorithms, the types of exchanges that accept algorithmic trading as well as more sophisticated strategies. As with Harris' book above, it explains in detail how electronic trading markets work, the knowledge of which I also believe is an essential prerequisite for carrying out systematic strategies.

Quantitative Trading

The next set of books are about algorithmic/quantitative trading directly. They outline some of the basic concepts and describe particular strategies that can be implemented.

- **Quantitative Trading: How to Build Your Own Algorithmic Trading Business** by Ernest Chan[5] - Ernest Chan's first book is a "beginner's guide" to quantitative trading strategies. While it is not heavy on strategy ideas, it does present a framework for how to setup a trading business, with risk management ideas and implementation tools. This is a great book if you are completely new to algorithmic trading. The book makes use of MATLAB.
- **Algorithmic Trading: Winning Strategies and Their Rationale** by Ernest Chan[6] - Chan's second book is very heavy on strategy ideas. It essentially begins where the previous book left off and is updated to reflect "current market conditions". The book discusses mean reversion and momentum based strategies at the interday and intraday frequencies. it also briefly touches on high-frequency trading. As with the prior book it makes extensive use of MATLAB code.
- **Inside The Black Box: The Simple Truth About Quantitative and High-Frequency Trading, 2nd Ed** by Rishi Narang[12] - Narang's book provides an overview of the components of a trading system employed by a quantitative hedge fund, including alpha generators, risk management, portfolio optimisation and transaction costs. The second edition goes into significant detail on high-frequency trading techniques.
- **Volatility Trading** by Euan Sinclair[16] - Sinclair's book concentrates solely on volatility modelling/forecasting and options strategies designed to take advantage of these models. If you plan to trade options in a quantitative fashion then this book will provide many research ideas.

5.2.2 The Internet

After gaining a grounding in the process of algorithmic trading via the classic texts, additional strategy ideas can be sourced from the internet. Quantitative finance blogs, link aggregators and trading forums all provide rich sources of ideas to test.

However, a note of caution: Many internet trading resources rely on the concept of **technical analysis**. Technical analysis involves utilising basic signals analysis *indicators* and *behavioural psychology* to determine trends or reversal patterns in asset prices.

Despite being extremely popular in the overall trading space, technical analysis is considered somewhat controversial in the quantitative finance community. Some have suggested that it is no better than reading a horoscope or studying tea leaves in terms of its predictive power! In reality there are successful individuals making extensive use of technical analysis in their trading.

As quants with a more sophisticated mathematical and statistical toolbox at our disposal, we can easily evaluate the effectiveness of such "TA-based" strategies. This allows us to make decisions driven by data analysis and hypothesis testing, rather than base such decisions on emotional considerations or preconceptions.

Quant Blogs

I recommend the following quant blogs for good trading ideas and concepts about algorithmic trading in general:

- **MATLAB Trading** - http://matlab-trading.blogspot.co.uk/
- **Quantitative Trading (Ernest Chan)** - http://epchan.blogspot.com
- **Quantivity** - http://quantivity.wordpress.com
- **Quantopian** - http://blog.quantopian.com
- **Quantpedia** - http://quantpedia.com

Aggregators

It has become fashionable in the last few years for topical links to be aggregated and then discussed. I read the following aggregators:

- **Quantocracy** - http://www.quantocracy.com
- **Quant News** - http://www.quantnews.com
- **Algo Trading Sub-Reddit** - http://www.reddit.com/r/algotrading

Forums

The next place to find additional strategy ideas is with trading forums. Do not be put off by more "technical analysis" oriented strategies. These strategies often provide good ideas that can be statistically tested:

- **Elite Trader Forums** - http://www.elitetrader.com
- **Nuclear Phynance** - http://www.nuclearphynance.com
- **QuantNet** - http://www.quantnet.com
- **Wealth Lab** - http://www.wealth-lab.com/Forum
- **Wilmott Forums** - http://www.wilmott.com

5.2.3 Journal Literature

Once you have had some experience at evaluating simpler strategies, it is time to look at the more sophisticated academic offerings. Some academic journals will be difficult to access, without high subscriptions or one-off costs. If you are a member or alumnus of a university, you should be able to obtain access to some of these financial journals. Otherwise, you can look at *pre-print servers*, which are internet repositories of late drafts of academic papers that are undergoing peer review. Since we are only interested in strategies that we can successfully replicate, backtest and obtain profitability for, a peer review is of less importance to us.

The major downside of academic strategies is that they can often either be out of date, require obscure and expensive historical data, trade in illiquid asset classes or do not factor in fees, slippage or spread. It can also be unclear whether the trading strategy is to be carried out with market orders, limit orders or whether it contains stop losses etc. Thus it is absolutely essential to replicate the strategy yourself as best you can, backtest it and add in realistic transaction costs that include as many aspects of the asset classes that you wish to trade in.

Here is a list of the more popular pre-print servers and financial journals that you can source ideas from:

- **arXiv** - http://arxiv.org/archive/q-fin
- **SSRN** - http://www.ssrn.com
- **Journal of Investment Strategies** - http://www.risk.net/type/journal/source/journal-of-investment-strategies
- **Journal of Computational Finance** - http://www.risk.net/type/journal/source/journal-of-computational-finance
- **Mathematical Finance** - http://onlinelibrary.wiley.com/journal/10.1111/%28ISSN%291467-9965

5.2.4 Independent Research

What about forming your own quantitative strategies? This generally requires (but is not limited to) expertise in one or more of the following categories:

- **Market microstructure** - For higher frequency strategies in particular, one can make use of *market microstructure*, i.e. understanding of the *order book dynamics* in order to generate profitability. Different markets will have various technology limitations, regulations, market participants and constraints that are all open to exploitation via specific strategies. This is a very sophisticated area and retail practitioners will find it hard to be competitive in this space, particularly as the competition includes large, well-capitalised quantitative hedge funds with strong technological capabilities.
- **Fund structure** - Pooled investment funds, such as pension funds, private investment partnerships (hedge funds), commodity trading advisors and mutual funds are constrained both by heavy regulation and their large capital reserves. Thus certain consistent behaviours can be exploited with those who are more nimble. For instance, large funds are subject to *capacity constraints* due to their size. Thus if they need to rapidly offload (sell) a quantity of securities, they will have to stagger it in order to avoid "moving the market". Sophisticated algorithms can take advantage of this, and other idiosyncrasies, in a general process known as *fund structure arbitrage.*
- **Machine learning/artificial intelligence** - Machine learning algorithms have become more prevalent in recent years in financial markets. Classifiers (such as Naive-Bayes, et al.) non-linear function matchers (neural networks) and optimisation routines (genetic algorithms) have all been used to predict asset paths or optimise trading strategies. If you have a background in this area you may have some insight into how particular algorithms might be applied to certain markets.

By continuing to monitor the above sources on a weekly, or even daily, basis you are setting yourself up to receive a consistent list of strategies from a diverse range of sources. The next step is to determine how to reject a large subset of these strategies in order to minimise wasting your time and backtesting resources on strategies that are likely to be unprofitable.

5.3 Evaluating Trading Strategies

The first, and arguably most obvious consideration is whether you actually **understand the strategy**. Would you be able to explain the strategy concisely or does it require a string of caveats and endless parameter lists? In addition, does the strategy have a good, solid basis in reality? For instance, could you point to some behavioural rationale or fund structure constraint that might be causing the pattern(s) you are attempting to exploit? Would this constraint hold up to a regime change, such as a dramatic regulatory environment disruption? Does the strategy rely on complex statistical or mathematical rules? Does it apply to any financial time series or is it specific to the asset class that it is claimed to be profitable on? You should constantly be thinking about these factors when evaluating new trading methods, otherwise you may waste a significant amount of time attempting to backtest and optimise unprofitable strategies.

Once you have determined that you understand the basic principles of the strategy you need to decide whether it fits with your aforementioned personality profile. This is not as vague a consideration as it sounds! Strategies will differ substantially in their performance characteristics. There are certain personality types that can handle more significant periods of drawdown, or are willing to accept greater risk for larger return. Despite the fact that we, as quants, try and eliminate as much cognitive bias as possible and should be able to evaluate a strategy dispassionately, biases will always creep in. Thus we need a consistent, unemotional means through which to assess the performance of strategies. Here is the list of criteria that I judge a potential new strategy by:

- **Methodology** - Is the strategy momentum based, mean-reverting, market-neutral, directional? Does the strategy rely on sophisticated (or complex!) statistical or machine learning techniques that are hard to understand and require a PhD in statistics to grasp? Do these techniques introduce a significant quantity of parameters, which might lead to optimisation bias? Is the strategy likely to withstand a *regime change* (i.e. potential new regulation of financial markets)?

- **Sharpe Ratio** - The Sharpe ratio heuristically characterises the reward/risk ratio of the strategy. It quantifies how much return you can achieve for the level of volatility endured by the equity curve. Naturally, we need to determine the period and frequency that these returns and volatility (i.e. standard deviation) are measured over. A higher frequency strategy will require greater sampling rate of standard deviation, but a shorter overall time period of measurement, for instance.

- **Leverage** - Does the strategy require significant leverage in order to be profitable? Does the strategy necessitate the use of leveraged derivatives contracts (futures, options, swaps) in order to make a return? These leveraged contracts can have heavy volatility characterises and thus can easily lead to *margin calls*. Do you have the trading capital and the temperament for such volatility?

- **Frequency** - The frequency of the strategy is intimately linked to your technology stack (and thus technological expertise), the Sharpe ratio and overall level of transaction costs. All other issues considered, higher frequency strategies require more capital, are more sophisticated and harder to implement. However, assuming your backtesting engine is sophisticated and bug-free, they will often have far higher Sharpe ratios.

- **Volatility** - Volatility is related strongly to the "risk" of the strategy. The Sharpe ratio characterises this. Higher volatility of the underlying asset classes, if unhedged, often leads to higher volatility in the equity curve and thus smaller Sharpe ratios. I am of course assuming that the positive volatility is approximately equal to the negative volatility. Some strategies may have greater downside volatility. You need to be aware of these attributes.

- **Win/Loss, Average Profit/Loss** - Strategies will differ in their win/loss and average profit/loss characteristics. One can have a very profitable strategy, even if the number of losing trades exceed the number of winning trades. Momentum strategies tend to have this pattern as they rely on a small number of "big hits" in order to be profitable. Mean-reversion strategies tend to have opposing profiles where more of the trades are "winners", but the losing trades can be quite severe.
- **Maximum Drawdown** - The maximum drawdown is the largest overall peak-to-trough percentage drop on the equity curve of the strategy. Momentum strategies are well known to suffer from periods of extended drawdowns (due to a string of many incremental losing trades). Many traders will give up in periods of extended drawdown, even if historical testing has suggested this is "business as usual" for the strategy. You will need to determine what percentage of drawdown (and over what time period) you can accept before you cease trading your strategy. This is a highly personal decision and thus must be considered carefully.
- **Capacity/Liquidity** - At the retail level, unless you are trading in a highly illiquid instrument (like a small-cap stock), you will not have to concern yourself greatly with strategy *capacity*. Capacity determines the scalability of the strategy to further capital. Many of the larger hedge funds suffer from significant capacity problems as their strategies increase in capital allocation.
- **Parameters** - Certain strategies (especially those found in the machine learning community) require a large quantity of parameters. Every extra parameter that a strategy requires leaves it more vulnerable to optimisation bias (also known as "curve-fitting"). You should try and target strategies with as few parameters as possible or make sure you have sufficient quantities of data with which to test your strategies on.
- **Benchmark** - Nearly all strategies (unless characterised as "absolute return") are measured against some performance benchmark. The benchmark is usually an *index* that characterises a large sample of the underlying asset class that the strategy trades in. If the strategy trades large-cap US equities, then the S&P500 would be a natural benchmark to measure your strategy against. You will hear the terms "alpha" and "beta", applied to strategies of this type.

Notice that we have not discussed the actual *returns* of the strategy. Why is this? In isolation, the returns actually provide us with limited information as to the effectiveness of the strategy. They don't give you an insight into leverage, volatility, benchmarks or capital requirements. Thus strategies are rarely judged on their returns alone. Always consider the risk attributes of a strategy before looking at the returns.

At this stage many of the strategies found from your pipeline will be rejected out of hand, since they won't meet your capital requirements, leverage constraints, maximum drawdown tolerance or volatility preferences. The strategies that do remain can now be considered for *backtesting*. However, before this is possible, it is necessary to consider one final rejection criteria - that of available historical data on which to test these strategies.

5.4 Obtaining Historical Data

Nowadays, the breadth of the technical requirements across asset classes for historical data storage is substantial. In order to remain competitive, both the buy-side (funds, prop-desks) and sell-side (broker/dealers) invest heavily in their technical infrastructure. It is imperative to consider its importance. In particular, we are interested in timeliness, accuracy and storage requirements. We will be discussing data storage in later chapters of the book.

In the previous section we had set up a strategy pipeline that allowed us to reject certain strategies based on our own personal rejection criteria. In this section we will filter more strategies based on our own preferences for obtaining historical data. The chief considerations (especially at retail practitioner level) are the costs of the data, the storage requirements and your level of technical expertise. We also need to discuss the different types of available data and the different considerations that each type of data will impose on us.

Let's begin by discussing the types of data available and the key issues we will need to think about, with the understanding that we will explore these issues in significant depth in the remainder of the book:

- **Fundamental Data** - This includes data about macroeconomic trends, such as interest rates, inflation figures, corporate actions (dividends, stock-splits), SEC filings, corporate accounts, earnings figures, crop reports, meteorological data etc. This data is often used to value companies or other assets on a *fundamental* basis, i.e. via some means of expected future cash flows. It does not include stock price series. Some fundamental data is freely available from government websites. Other long-term historical fundamental data can be extremely expensive. Storage requirements are often not particularly large, unless thousands of companies are being studied at once.

- **News Data** - News data is often qualitative in nature. It consists of articles, blog posts, microblog posts ("tweets") and editorial. Machine learning techniques such as *classifiers* are often used to interpret *sentiment*. This data is also often freely available or cheap, via subscription to media outlets. The newer "NoSQL" document storage databases are designed to store this type of unstructured, qualitative data.

- **Asset Price Data** - This is the traditional data domain of the quant. It consists of time series of asset prices. Equities (stocks), fixed income products (bonds), commodities and foreign exchange prices all sit within this class. Daily historical data is often straightforward to obtain for the simpler asset classes, such as equities. However, once accuracy and cleanliness are included and statistical biases removed, the data can become expensive. In addition, time series data often possesses significant storage requirements especially when intraday data is considered.

- **Financial Instruments** - Equities, bonds, futures and the more exotic derivative options have very different characteristics and parameters. Thus there is no "one size fits all" database structure that can accommodate them. Significant care must be given to the design and implementation of database structures for various financial instruments.

- **Frequency** - The higher the frequency of the data, the greater the costs and storage requirements. For low-frequency strategies, daily data is often sufficient. For high frequency strategies, it might be necessary to obtain tick-level data and even historical copies of particular trading exchange *order book* data. Implementing a storage engine for this type of data is very technologically intensive and only suitable for those with a strong programming/technical background.

- **Benchmarks** - The strategies described above will often be compared to a *benchmark*. This usually manifests itself as an additional financial time series. For equities, this is often a national stock benchmark, such as the S&P500 index (US) or FTSE100 (UK). For a fixed income fund, it is useful to compare against a basket of bonds or fixed income products. The "risk-free rate" (i.e. appropriate interest rate) is also another widely accepted benchmark. All asset class categories possess a favoured benchmark, so it will be necessary to research this based on your particular strategy, if you wish to gain interest in your strategy externally.

- **Technology** - The technology stacks behind a financial data storage centre are complex. However, it does generally centre around a database cluster engine, such as a Relational Database Management System (RDBMS), such as MySQL, SQL Server, Oracle or a Document Storage Engine (i.e. "NoSQL"). This is accessed via "business logic" application code that queries the database and provides access to external tools, such as MATLAB, R or Excel. Often this business logic is written in C++, Java or Python. You will also need to host this data somewhere, either on your own personal computer, or remotely via internet servers. Products such as Amazon Web Services have made this simpler and cheaper in recent years, but it will still require significant technical expertise to achieve in a robust manner.

As can be seen, once a strategy has been identified via the pipeline it will be necessary to evaluate the availability, costs, complexity and implementation details of a particular set of historical data. You may find it is necessary to reject a strategy based solely on historical data considerations. This is a big area and teams of PhDs work at large funds making sure pricing is accurate and timely. Do not underestimate the difficulties of creating a robust data centre for your backtesting purposes!

I do want to say, however, that many backtesting platforms can provide this data for you automatically - at a cost. Thus it will take much of the implementation pain away from you, and you can concentrate purely on strategy implementation and optimisation. Tools like TradeStation possess this capability. However, my personal view is to implement as much as possible internally and avoid outsourcing parts of the stack to software vendors. I prefer higher frequency strategies due to their more attractive Sharpe ratios, but they are often tightly coupled to the technology stack, where advanced optimisation is critical.

Part III

Data Platform Development

Chapter 6

Software Installation

This chapter will discuss in detail how to install an algorithmic trading environment. Operating system choice is considered as a necessary first step, with the three major choices outlined. Subsequently Linux is chosen as the system of choice (Ubuntu in particular) and Python is installed with all of the necessary libraries.

Package/library installation is often glossed over in additional books but I personally feel that it can be a stumbling block for many so I have devoted an entire chapter to it. Unfortunately the reality is that the chapter will become dated the moment it is released. Newer versions of operating systems emerge and packages are constantly updated. Hence there are likely to be specific implementation details.

If you do have trouble installing or working with these packages, make sure to check the versions installed and upgrade if necessary. If you still have trouble, feel free to email me at mike@quantstart.com and I'll try and help you out.

6.1 Operating System Choice

The first major choice when deciding on an algorithmic trading platform is that of the operating system. In some sense this will be dictated by the primary programming language or the means of connecting to the brokerage. These days the majority of software, particularly open source, is cross-platform and so the choice is less restricted.

6.1.1 Microsoft Windows

Windows is probably the "default" option of many algorithmic traders. It is extremely familiar and, despite criticism to the contrary, in certain forms is rather robust. Windows 8 has not been hugely well received but the prior version, Windows 7, is considered a solid operating system.

Certain tools in the algorithmic trading space will only function on Windows, in particular the IQFeed server, necessary to download tick data from DTN IQFeed. In addition Windows is the native platform of the Microsoft .NET framework, on which a vast quantity of financial software is written, utilising C++ and C#.

If you do not wish to use Windows then it is sometimes possible to run Windows-based software under a UNIX based system using the WINE emulator (http://www.winehq.org/).

6.1.2 Mac OSX

Mac OSX combines the graphical ease of Windows (some say it improves substantially upon it!) with the robustness of a UNIX based system (FreeBSD). While I use a MacBook Air for all of my "day to day" work, such as web/email and developing the QuantStart site, I have found it to be extremely painful to install a full algorithmic research stack, based on Python, under Mac OSX.

The package landscape of Mac OSX is significantly fragmented, with Homebrew and MacPorts being the primary contenders. Installation from source is tricky due to the proprietary compilation process (using XCode). I have not yet successfully installed NumPy, SciPy and pandas on my MacBook as of this writing!

However, if you can navigate the minefield that is Python installation on Mac OSX, it can provide a great environment for algorithmic research. Since the Interactive Brokers Trader Workstation is Java-based, it has no trouble running on Mac OSX.

6.1.3 Linux

"Linux" refers to a set of free UNIX distributions such as Cent OS, Debian and Ubuntu. I don't wish to go into details about the benefits/drawbacks of each distribution, rather I will concentrate on Debian-based distro. In particular I will be considering Ubuntu Desktop as the algorithmic trading environment.

The aptitude package management makes it straightforward to install the necessary underlying libraries with ease. In addition it is straightforward to create a *virtual environment* for Python that can isolate your algo trading code from other Python apps. I have never had any (major) trouble installing a Python environment on a modern Ubuntu system and as such I have chosen this as the primary environment from which to conduct my trading.

If you would like to give Ubuntu a go before committing fully, by dual-booting for example, then it is possible to use VirtualBox (https://www.virtualbox.org/) to install it. I have a detailed guide on QuantStart (http://www.quantstart.com/articles/Installing-a-Desktop-Algorithmic-Trading-Research-Environment-using-Ubuntu-Linux-and-Python), which describes the process.

6.2 Installing a Python Environment on Ubuntu Linux

In this section we will discuss how to set up a robust, efficient and interactive development environment for algorithmic trading strategy research making use of Ubuntu Desktop Linux and the Python programming language. We will utilise this environment for all subsequent algorithmic trading implementations.

To create the research environment we will install the following software tools, all of which are open-source and free to download:

- **Ubuntu Desktop Linux** - The operating system
- **Python** - The core programming environment
- **NumPy/SciPy** - For fast, efficient vectorised array/matrix calculation
- **IPython** - For visual interactive development with Python
- **matplotlib** - For graphical visualisation of data
- **pandas** - For data "wrangling" and time series analysis
- **scikit-learn** - For machine learning and artificial intelligence algorithms
- **IbPy** - To carry out trading with the Interactive Brokers API

These tools coupled with a suitable MySQL securities master database will allow us to create a rapid interactive strategy research and backtesting environment. Pandas is designed for "data wrangling" and can import and cleanse time series data very efficiently. NumPy/SciPy running underneath keeps the system extremely well optimised. IPython/matplotlib (and the qtconsole described below) allow interactive visualisation of results and rapid iteration. scikit-learn allows us to apply machine learning techniques to our strategies to further enhance performance.

6.2.1 Python

The latest versions of Ubuntu, which at the time of writing is 13.10, still make use of the Python 2.7.x version family. While there is a transition underway to 3.3.x the majority of libraries are fully compatible with the 2.7.x branch. Thus I have chosen to use this for algorithmic trading. Things are likely to evolve rapidly though so in a couple of years 3.3.x may be the predominant branch. We will now commence with the installation of the Python environment.

The first thing to do on any brand new Ubuntu Linux system is to *update* and *upgrade* the packages. The former tells Ubuntu about new packages that are available, while the latter actually performs the process of replacing older packages with newer versions. Run the following commands in a terminal session and you will be prompted for your passwords:

```
sudo apt-get -y update
sudo apt-get -y upgrade
```

Note that the -y prefix tells Ubuntu that you want to accept 'yes' to all yes/no questions. "sudo" is a Ubuntu/Debian Linux command that allows other commands to be executed with administrator privileges. Since we are installing our packages sitewide, we need 'root access' to the machine and thus must make use of 'sudo'.

Once both of those updating commands have been successfully executed we need to install the Python development packages and compilers needed to compile all of the software. Notice that we are installing `build-essential` which contains the GCC compilers and the LAPACK linear algebra library, as well as **pip** which is the Python package management system:

```
sudo apt-get install python-pip python-dev python2.7-dev \
build-essential liblapack-dev libblas-dev
```

The next stage is to install the Python numerical and data analysis libraries.

6.2.2 NumPy, SciPy and Pandas

Once the necessary packages are installed above we can go ahead and install NumPy via pip, the Python package manager. Pip will download a zip file of the package and then compile it from the source code for us. Bear in mind that it will take some time to compile, possible 10 minutes or longer depending upon your CPU:

```
sudo pip install numpy
```

Once NumPy has been installed we need to check that it works before proceeding. If you look in the terminal you'll see your username followed by your computer name. In my case it is `mhallsmoore@algobox`, which is followed by the prompt. At the prompt type `python` and then try importing NumPy. We will test that it works by calculating the mean average of a list:

```
mhallsmoore@algobox:~$ python
Python 2.7.4 (default, Sep 26 2013, 03:20:26)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> from numpy import mean
>>> mean([1,2,3])
2.0
>>> exit()
```

Now that NumPy has been successfully installed we want to install the Python Scientific library known as SciPy. It has a few package dependencies of its own including the ATLAS library and the GNU Fortran compiler, which must be installed first:

```
sudo apt-get install libatlas-base-dev gfortran
```

We are ready to install SciPy now, with pip. This will take quite a long time to compile, perhaps 10-20 minutes, depending upon CPU speed:

```
sudo pip install scipy
```

SciPy has now been installed. We will test it out in a similar fashion to NumPy when calculating the standard deviation of a list of integers:

```
mhallsmoore@algobox:~$ python
Python 2.7.4 (default, Sep 26 2013, 03:20:26)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import scipy
>>> from scipy import std
>>> std([1,2,3])
0.81649658092772603
>>> exit()
```

The final task for this section is to install the pandas data analysis library. We don't need any additional dependencies at this stage as they're covered by NumPy and SciPy:

```
sudo pip install pandas
```

We can now test the pandas installation, as before:

```
>>> from pandas import DataFrame
>>> pd = DataFrame()
>>> pd
Empty DataFrame
Columns: []
Index: []
>>> exit()
```

Now that the base numerical and scientific libraries have been installed we will install the statistical and machine learning libraries, statsmodels and scikit-learn.

6.2.3 Statsmodels and Scikit-Learn

Installation proceeds as before, making use of pip to install the packages:

```
sudo pip install statsmodels
sudo pip install scikit-learn
```

Both libraries can be tested:

```
mhallsmoore@algobox:~$ python
Python 2.7.4 (default, Sep 26 2013, 03:20:26)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> iris
..
..
'petal width (cm)']}
>>>
```

Now that the two statistical libraries are installed we can install the visualisation and development tools, IPython and matplotlib.

6.2.4 PyQt, IPython and Matplotlib

The first task is to install the dependency packages for matplotlib, the Python graphing library. Since matplotlib is a Python package, we cannot use pip to install the underlying libraries for working with PNGs, JPEGs and freetype fonts, so we need Ubuntu to install them for us:

```
sudo apt-get install libpng-dev libjpeg8-dev libfreetype6-dev
```

Now we can install matplotlib:

```
sudo pip install matplotlib
```

The last task of this section is to instal IPython. This is an interactive Python interpreter that provides a significantly more streamlined workflow compared to using the standard Python console. In later chapters we will emphasise the full usefulness of IPython for algorithmic trading development:

```
sudo pip install ipython
```

While IPython is sufficiently useful on its own, it can be made even more powerful by including the *qtconsole*, which provides the ability to inline matplotlib visualisations. However, it takes a little bit more work to get this up and running.

First, we need to install the the **Qt library**:

```
sudo apt-get install libqt4-core libqt4-gui libqt4-dev
```

The qtconsole has a few additional dependency packages, namely the ZMQ and Pygments libraries:

```
sudo apt-get install libzmq-dev
sudo pip install pyzmq
sudo pip install pygments
```

It is straightforward to test IPython by typing the following command:

```
ipython qtconsole --pylab=inline
```

To test IPython a simple plot can be generated by typing the following commands. Note that I've included the IPython numbered input/outut which you do not need to type:

```
In [1]: x=np.array([1,2,3])

In [2]: plot(x)
Out[2]: [<matplotlib.lines.Line2D at 0x392a1d0>]
```

This should display an inline matplotlib graph. Closing IPython allows us to continue with the installation.

6.2.5 IbPy and Trader Workstation

Interactive Brokers is one of the main brokerages used by retail algorithmic traders due to its relatively low minimal account balance requirements (10,000 USD) and (relatively) straightforward API. In this section we will install IbPy and Trader Workstation, which we will later use to carry out automated trade execution.

I want to emphasise that we are not going to be trading any live capital with this download! We are simply going to be installing some software which will let us try out a "demo account", which provides a market simulator with out of date data in a "real time" fashion.

Disclosure: I have no affiliation with Interactive Brokers. I have used them before in a professional fund context and as such am familiar with their software.

IbPy is a Python wrapper written around the Java-based Interactive Brokers API. It makes development of algorithmic trading systems in Python somewhat less problematic. It will be used as the basis for all subsequent communication with Interactive Brokers. An alternative is to use the FIX protocol, but we won't consider that method in this book.

Since IBPy is maintained on the GitHub source code version control website, as a git repository, we will need to install git. This is handled by:

```
sudo apt-get install git-core
```

Once git has been installed it is necessary to create a subdirectory to store IBPy. It can simply be placed underneath the home directory:

```
mkdir ~/ibapi
```

The next step is to download IBPy via the 'git clone' command:

```
cd ~/ibapi
git clone https://github.com/blampe/IbPy
```

The final step is to enter the IbPy directory and install using Python setuptools:

```
cd ~/ibapi/IbPy
python setup.py.in install
```

That completes the installation of IBPy. The next step is to install Trader Workstation. At the time of writing, it was necessary to follow this link (IB), which takes you directly to the Trader Workstation download page at Interactive Brokers. Select the platform that you wish to utilise. In this instance I have chosen the UNIX download, which can be found here (IB Unix Download).

At that link it will describe the remainder of the process but I will replicate it here for completeness. The downloaded file will be called *unixmacosx_ latest.jar*. Open the file:

```
jar xf unixmacosx_latest.jar
```

Then change to the IBJts directory and load TWS:

```
cd IBJts
java -cp jts.jar:total.2013.jar -Xmx512M -XX:MaxPermSize=128M jclient.LoginFrame .
```

This will present you with the Trader Workstation login screen. If you choose the username "edemo" and the password "demo user" you will be logged into the system.

This completes the installation of a full algorithmic trading environment under Python and Ubuntu. The next stage is to begin collecting and storing historical pricing data for our strategies.

Chapter 7

Financial Data Storage

In algorithmic trading the spotlight usually shines on the *alpha model* component of the full trading system. This component generates the trading signals, prior to filtration by a risk management and portfolio construction system. As such, algo traders often spend a significant portion of their research time refining the alpha model in order to optimise one or more metrics prior to deployment of the strategy into production.

However, an alpha model is only as good as the data being fed into it. This concept is nicely characterised by the old computer science adage of *"garbage in, garbage out."* It is absolutely crucial that accurate, timely data is used to feed the alpha model. Otherwise results will be at best poor or at worst completely incorrect. This will lead to significant underperformance when system is deployed live.

In this chapter we will discuss issues surrounding the acquisition and provision of timely accurate data for an algorithmic strategy backtesting system and ultimately a trading execution engine. In particular we will study how to obtain financial data and how to store it. Subsequent chapters will discuss how to clean it and how to export it. In the financial industry this type of data service is known as a **securities master database**.

7.1 Securities Master Databases

A securities master is an organisation-wide database that stores **fundamental**, **pricing** and **transactional** data for a variety of financial instruments across asset classes. It provides access to this information in a consistent manner to be used by other departments such as risk management, clearing/settlement and proprietary trading.

In large organisations a range of instruments and data will be stored. Here are some of the instruments that might be of interest to a firm:

- Equities
- Equity Options
- Indices
- Foreign Exchange
- Interest Rates
- Futures
- Commodities
- Bonds - Government and Corporate
- Derivatives - Caps, Floors, Swaps

Securities master databases often have teams of developers and data specialists ensuring *high availability* within a financial institution. While this is necessary in large companies, at the retail level or in a small fund a securities master can be far simpler. In fact, while large securities masters make use of expensive enterprise database and analysis systems, it is possibly to use commodity open-source software to provide the same level of functionality, assuming a well-optimised system.

7.2 Financial Datasets

For the retail algorithmic trader or small quantitative fund the most common data sets are end-of-day and intraday historical pricing for equities, indices, futures (mainly commodities or fixed income) and foreign exchange (forex). In order to simplify this discussion we will concentrate solely on end-of-day (EOD) data for equities, ETFs and equity indices. Later sections will discuss adding higher frequency data, additional asset classes and derivatives data, which have more advanced requirements.

EOD data for equities is easy to obtain. There are a number of services providing access for free via web-available APIs:

- **Yahoo Finance** - http://finance.yahoo.com
- **Google Finance** - https://www.google.com/finance
- **QuantQuote** - https://www.quantquote.com (S&P500 EOD data only)
- **EODData** - http://eoddata.com (requires registration)

It is straightforward to manually download historical data for individual securities but it becomes time-consuming if many stocks need to be downloaded daily. Thus an important component of our securities master will be automatically updating the data set.

Another issue is *look-back period*. How far in the past do we need to go with our data? This will be specific to the requirements of your trading strategy, but there are certain problems that span all strategies. The most common is **regime change**, which is often characterised by a new regulatory environment, periods of higher/lower volatility or longer-term trending markets. For instance a long-term short-directional trend-following/momentum strategy would likely perform very well from 2000-2003 or 2007-2009. However it would have had a tough time from 2003-2007 or 2009 to the present.

My rule of thumb is to obtain as much data as possible, especially for EOD data where storage is cheap. Just because the data exists in your security master, does not mean it must be utilised. There are caveats around performance, as larger database tables mean longer query times (see below), but the benefits of having more sample points generally outweighs any performance issues.

As with all financial data it is imperative to be aware of errors, such as incorrect high/low prices or **survivorship bias**, which I have discussed at length in previous chapters.

7.3 Storage Formats

There are three main ways to store financial data. They all possess varying degrees of access, performance and structural capabilities. We will consider each in turn.

7.3.1 Flat-File Storage

The simplest data store for financial data, and the way in which you are likely to receive the data from any data vendors, is the flat-file format. Flat-files often make use of the Comma-Separated Variable (CSV) format, which store a two-dimensional matrix of data as a series of rows, with column data separated via a delimiter (often a comma, but can be whitespace, such as a space or tab). For EOD pricing data, each row represents a trading day via the OHLC paradigm (i.e. the prices at the open, high, low and close of the trading period).

The advantage of flat-files are their simplicity and ability to be heavily compressed for archiving or download. The main disadvantages lie in their lack of query capability and poor performance for iteration across large datasets. **SQLite** and **Excel** mitigate some of these problems by providing certain querying capabilities.

7.3.2 Document Stores/NoSQL

Document stores/NoSQL databases, while certainly not a new concept, have gained significant prominence in recent years due to their use at "web-scale" firms such as Google, Facebook and Twitter. They differ substantially from RDBMS systems in that there is no concept of table schemas. Instead, there are *collections* and *documents*, which are the closest analogies to tables and records, respectively. A wide taxonomy of document stores exist, the discussion of which is well outside this chapter! However, some of the more popular stores include **MongoDB**, **Cassandra** and **CouchDB**.

Document stores, in financial applications, are mostly suited to fundamental or meta data. Fundamental data for financial assets comes in many forms, such as corporate actions, earnings statements, SEC filings etc. Thus the schema-less nature of NoSQL DBs is well-suited. However, NoSQL DBs are not well designed for time-series such as high-resolution pricing data and so we won't be considering them further in this chapter.

7.3.3 Relational Database Management Systems

A *relational database management system* (RDBMS) makes use of the *relational model* to store data. These databases are particular well-suited to financial data because different "objects" (such as exchanges, data sources, prices) can be separated into tables with relationships defined between them.

RDBMS make use of **Structured Query Language** (SQL) in order to perform complex data queries on financial data. Examples of RDBMS include **Oracle**, **MySQL**, **SQLServer** and **PostgreSQL**.

The main advantages of RDBMS are their simplicity of installation, platform-independence, ease of querying, ease of integration with major backtest software and high-performance capabilities at large scale (although some would argue the latter is not the case!). Their disadvantages are often due to the complexity of customisation and difficulties of achieving said performance without underlying knowledge of how RDBMS data is stored. In addition, they possess semi-rigid schemas and so data often has to be modified to fit into such designs. This is unlike NoSQL data stores, where there is no schema.

For all of the future historical pricing implementation code in the book we will make use of the MySQL RDBMS. It is free, open-source, cross-platform, highly robust and its behaviour at scale is well-documented, which makes it a sensible choice for quant work.

7.4 Historical Data Structure

There is a significant body of theory and academic research carried out in the realm of computer science for the optimal design for data stores. However, we won't be going into too much detail as it is easy to get lost in minutiae! Instead I will present a common pattern for the construction of an equities security master, which you can modify as you see fit for your own applications.

The first task is to define our *entities*, which are elements of the financial data that will eventually map to tables in the database. For an equities master database I foresee the following entities:

- **Exchanges** - What is the ultimate original source of the data?
- **Vendor** - Where is a particular data point obtained from?
- **Instrument/Ticker** - The ticker/symbol for the equity or index, along with corporate information of the underlying firm or fund.

- **Price** - The actual price for a particular security on a particular day.
- **Corporate Actions** - The list of all stock splits or dividend adjustments (this may lead to one or more tables), necessary for adjusting the pricing data.
- **National Holidays** - To avoid mis-classifying trading holidays as missing data errors, it can be useful to store national holidays and cross-reference.

There are significant issues with regards to storing canonical tickers. I can attest to this from first hand experience at a hedge fund dealing with this exact problem! Different vendors use different methods for resolving tickers and thus combining multiple sources for accuracy. Further, companies become bankrupt, are exposed to M&A activity (i.e. become acquired and change names/symbols) and can have multiple publicly traded share classes. Many of you will not have to worry about this because your universe of tickers will be limited to the larger index constituents (such as the S&P500 or FTSE350).

7.5 Data Accuracy Evaluation

Historical pricing data from vendors is prone to many forms of error:

- **Corporate Actions** - Incorrect handling of stock splits and dividend adjustments. One must be absolutely sure that the formulae have been implemented correctly.
- **Spikes** - Pricing points that greatly exceed certain historical volatility levels. One must be careful here as these spikes do occur - see the May Flash Crash for a scary example. Spikes can also be caused by not taking into account stock splits when they do occur. *Spike filter* scripts are used to notify traders of such situations.
- **OHLC Aggregation** - Free OHLC data, such as from Yahoo/Google is particular prone to 'bad tick aggregation' situations where smaller exchanges process small trades well above the 'main' exchange prices for the day, thus leading to over-inflated maxima/minima once aggregated. This is less an 'error' as such, but more of an issue to be wary of.
- **Missing Data** - Missing data can be caused by lack of trades in a particular time period (common in second/minute resolution data of illiquid small-caps), by trading holidays or simply an error in the exchange system. Missing data can be *padded* (i.e. filled with the previous value), *interpolated* (linearly or otherwise) or ignored, depending upon the trading system.

Many of these errors rely on manual judgement in order to decide how to proceed. It is possible to automate the notification of such errors, but it is much harder to automate their solution. For instance, one must choose the threshold for being told about spikes - how many standard deviations to use and over what look-back period? Too high a stdev will miss some spikes, but too low and many unusual news announcements will lead to false positives. All of these issues require advanced judgement from the quant trader.

It is also necessary to decide how to fix errors. Should errors be corrected as soon as they're known, and if so, should an audit trail be carried out? This will require an extra table in the DB. This brings us to the topic of back-filling, which is a particularly insidious issue for backtesting. It concerns automatic correction of bad data upstream. If your data vendor corrects a historical error, but a backtested trading strategy is in production based on research from their previous bad data then decisions need to be made regarding the strategy effectiveness. This can be somewhat mitigated by being fully aware of your strategy performance metrics (in particular the variance in your win/loss characteristics for each trade). Strategies should be chosen or designed such that a single data point cannot skew the performance of the strategy to any great extent.

7.6 Automation

The benefit of writing software scripts to carry out the download, storage and cleaning of the data is that scripts can be automated via tools provided by the operating system. In UNIX-based systems (such as Mac OSX or Linux), one can make use of the **crontab**, which is a continually running process that allows specific scripts to be executed at custom-defined times or regular periods. There is an equivalent process on MS Windows known as the Task Scheduler.

A production process, for instance, might automate the download all of the S&P500 end-of-day prices as soon as they're published via a data vendor. It will then automatically run the aforementioned missing data and spike filtration scripts, alerting the trader via email, SMS or some other form of notification. At this point any backtesting tools will automatically have access to recent data, without the trader having to lift a finger! Depending upon whether your trading system is located on a desktop or on a remote server you may choose however to have a semi-automated or fully-automated process for these tasks.

7.7 Data Availability

Once the data is automatically updated and residing in the RDBMS it is necessary to get it into the backtesting software. This process will be highly dependent upon how your database is installed and whether your trading system is local (i.e. on a desktop computer) or remote (such as with a co-located exchange server).

One of the most important considerations is to minimise excessive Input/Output (I/O) as this can be extremely expensive both in terms of time and money, assuming remote connections where bandwidth is costly. The best way to approach this problem is to only move data across a network connection that you need (via selective querying) or exporting and compressing the data.

Many RDBMS support *replication* technology, which allows a database to be cloned onto another remote system, usually with a degree of latency. Depending upon your setup and data quantity this may only be on the order of minutes or seconds. A simple approach is to replicate a remote database onto a local desktop. However, be warned that synchronisation issues are common and time consuming to fix!

7.8 MySQL for Securities Masters

Now that we have discussed the idea behind a security master database it's time to actually build one. For this we will make use of two open source technologies: the **MySQL** database and the **Python** programming language. At the end of this chapter you will have a fully fledged equities security master with which to conduct further data analysis for your quantitative trading research.

7.8.1 Installing MySQL

Installation of MySQL within Ubuntu is straightforward. Simply open up a terminal and type the following:

```
sudo apt-get install mysql-server
```

Eventually, you will be prompted for a root password. This is your primary administration password so do not forget it! Enter the password and the installation will proceed and finish.

7.8.2 Configuring MySQL

Now that MySQL is installed on your system we can create a new *database* and a *user* to interact with it. You will have been prompted for a root password on installation. To log on to MySQL from the command line use the following line and then enter your password:

```
mysql -u root -p
```

Once you have logged in to the MySQL you can create a new database called `securities_master` and then select it:

```
mysql> CREATE DATABASE securities_master;
mysql> USE securities_master;
```

Once you create a database it is necessary to add a new *user* to interact with the database. While you can use the `root` user, it is considered bad practice from a security point of view, as it grants too many permissions and can lead to a compromised system. On a local machine this is mostly irrelevant but in a remote production environment you will certainly need to create a user with reduced permissions. In this instance our user will be called `sec_user`. Remember to replace `password` with a secure password:

```
mysql> CREATE USER 'sec_user'@'localhost' IDENTIFIED BY 'password';
mysql> GRANT ALL PRIVILEGES ON securities_master.* TO 'sec_user'@'localhost';
mysql> FLUSH PRIVILEGES;
```

The above three lines create and authorise the user to use `securities_master` and apply those privileges. From now on any interaction that occurs with the database will make use of the `sec_user` user.

7.8.3 Schema Design for EOD Equities

We've now installed MySQL and have configured a user with which to interact with our database. At this stage we are ready to construct the necessary tables to hold our financial data. For a simple, straightforward equities master we will create four tables:

- **Exchange** - The exchange table lists the exchanges we wish to obtain equities pricing information from. In this instance it will almost exclusively be the New York Stock Exchange (NYSE) and the National Association of Securities Dealers Automated Quotations (NASDAQ).
- **DataVendor** - This table lists information about historical pricing data vendors. We will be using Yahoo Finance to source our end-of-day (EOD) data. By introducing this table, we make it straightforward to add more vendors if necessary, such as Google Finance.
- **Symbol** - The symbol table stores the list of ticker symbols and company information. Right now we will be avoiding issues such as differing share classes and multiple symbol names.
- **DailyPrice** - This table stores the daily pricing information for each security. It can become very large if many securities are added. Hence it is necessary to optimise it for performance.

MySQL is an extremely flexible database in that it allows you to customise how the data is stored in an underlying *storage engine*. The two primary contenders in MySQL are **MyISAM** and **InnoDB**. Although I won't go into the details of storage engines (of which there are many!), I will say that MyISAM is more useful for fast reading (such as querying across large amounts of price information), but it doesn't support transactions (necessary to fully *rollback* a multi-step operation that fails mid way through). InnoDB, while transaction safe, is slower for reads.

InnoDB also allows row-level locking when making writes, while MyISAM locks the entire table when writing to it. This can have performance issues when writing a lot of information to arbitrary points in the table (such as with UPDATE statements). This is a deep topic, so I will leave the discussion to another day!

We are going to use InnoDB as it is natively transaction safe and provides row-level locking. If we find that a table is slow to be read, we can create *indexes* as a first step and then change the underlying storage engine if performance is still an issue. All of our tables will use the UTF-8 character set, as we wish to support international exchanges.

Let's begin with the schema and `CREATE TABLE` SQL code for the `exchange` table. It stores the abbreviation and name of the exchange (i.e. NYSE - New York Stock Exchange) as well as the geographic location. It also supports a currency and a timezone offset from UTC. We also store a created and last updated date for our own internal purposes. Finally, we set the primary index key to be an auto-incrementing integer ID (which is sufficient to handle 2^{32} records):

```
CREATE TABLE ‘exchange‘ (
  ‘id‘ int NOT NULL AUTO_INCREMENT,
  ‘abbrev‘ varchar(32) NOT NULL,
  ‘name‘ varchar(255) NOT NULL,
  ‘city‘ varchar(255) NULL,
  ‘country‘ varchar(255) NULL,
  ‘currency‘ varchar(64) NULL,
  ‘timezone_offset‘ time NULL,
  ‘created_date‘ datetime NOT NULL,
  ‘last_updated_date‘ datetime NOT NULL,
  PRIMARY KEY (‘id‘)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

Here is the schema and `CREATE TABLE` SQL code for the `data_vendor` table. It stores the name, website and support email. In time we can add more useful information for the vendor, such as an API endpoint URL:

```
CREATE TABLE ‘data_vendor‘ (
  ‘id‘ int NOT NULL AUTO_INCREMENT,
  ‘name‘ varchar(64) NOT NULL,
  ‘website_url‘ varchar(255) NULL,
  ‘support_email‘ varchar(255) NULL,
  ‘created_date‘ datetime NOT NULL,
  ‘last_updated_date‘ datetime NOT NULL,
  PRIMARY KEY (‘id‘)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

Here is the schema and `CREATE TABLE` SQL code for the `symbol` table. It contains a foreign key link to an exchange (we will only be supporting exchange-traded instruments for the time being), a ticker symbol (e.g. GOOG), an instrument type (’stock’ or ’index’), the name of the stock or stock market index, an equities sector and a currency.

```
CREATE TABLE ‘symbol‘ (
  ‘id‘ int NOT NULL AUTO_INCREMENT,
  ‘exchange_id‘ int NULL,
  ‘ticker‘ varchar(32) NOT NULL,
  ‘instrument‘ varchar(64) NOT NULL,
  ‘name‘ varchar(255) NULL,
  ‘sector‘ varchar(255) NULL,
  ‘currency‘ varchar(32) NULL,
  ‘created_date‘ datetime NOT NULL,
  ‘last_updated_date‘ datetime NOT NULL,
  PRIMARY KEY (‘id‘),
  KEY ‘index_exchange_id‘ (‘exchange_id‘)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

Here is the schema and `CREATE TABLE` SQL code for the `daily_price` table. This table is where the historical pricing data is actually stored. We have prefixed the table name with `daily_` as we may wish to create minute or second resolution data in separate tables at a later date for higher frequency strategies. The table contains two foreign keys - one to the data vendor and another to a symbol. This uniquely identifies the data point and allows us to store the same price data for multiple vendors in the same table. We also store a price date (i.e. the daily period over which the OHLC data is valid) and the created and last updated dates for our own purposes.

The remaining fields store the open-high-low-close and adjusted close prices. Yahoo Finance provides dividend and stock splits for us, the price of which ends up in the `adj_close_price` column. Notice that the datatype is `decimal(19,4)`. When dealing with financial data it is absolutely necessary to be precise. If we had used the `float` datatype we would end up with rounding errors due to the nature of how `float` data is stored internally. The final field stores the trading volume for the day. This uses the `bigint` datatype so that we don't accidentally truncate extremely high volume days.

```
CREATE TABLE ‘daily_price‘ (
  ‘id‘ int NOT NULL AUTO_INCREMENT,
  ‘data_vendor_id‘ int NOT NULL,
  ‘symbol_id‘ int NOT NULL,
  ‘price_date‘ datetime NOT NULL,
  ‘created_date‘ datetime NOT NULL,
  ‘last_updated_date‘ datetime NOT NULL,
  ‘open_price‘ decimal(19,4) NULL,
  ‘high_price‘ decimal(19,4) NULL,
  ‘low_price‘ decimal(19,4) NULL,
  ‘close_price‘ decimal(19,4) NULL,
  ‘adj_close_price‘ decimal(19,4) NULL,
  ‘volume‘ bigint NULL,
  PRIMARY KEY (‘id‘),
  KEY ‘index_data_vendor_id‘ (‘data_vendor_id‘),
  KEY ‘index_symbol_id‘ (‘symbol_id‘)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

By entering all of the above SQL commands into the MySQL command line the four necessary tables will be created.

7.8.4 Connecting to the Database

Before we can use MySQL with Python we need to install the `mysqlclient` library. `mysqlclient` is actually a fork of another library, known as Python-MySQL. Unfortunately the latter library is not supported in Python3 and so we must use `mysqlclient`. On Mac OSX/UNIX flavour machines we need to run the following commands:

```
sudo apt-get install libmysqlclient-dev
pip install mysqlclient
```

We're now ready to begin interacting with our MySQL database via Python and pandas.

7.8.5 Using an Object-Relational Mapper

For those of you with a background in database administration and development you might be asking whether it is more sensible to make use of an **Object-Relational Mapper** (ORM). An ORM allows objects within a programming language to be directly mapped to tables in databases such that the program code is fully unaware of the underlying storage engine. They are not without their problems, but they can save a great deal of time. The time-saving usually comes at the expense of performance, however.

A popular ORM for Python is **SQLAlchemy**. It allows you to specify the database schema within Python itself and thus automatically generate the `CREATE TABLE` code. Since we have specifically chosen MySQL and are concerned with performance, I've opted not to use an ORM for this chapter.

Symbol Retrieval

Let's begin by obtaining all of the ticker symbols associated with the Standard & Poor's list of 500 large-cap stocks, i.e. the S&P500. Of course, this is simply an example. If you are trading from the UK and wish to use UK domestic indices, you could equally well obtain the list of FTSE100 companies traded on the London Stock Exchange (LSE).

Wikipedia conveniently lists the constituents of the S&P500. Note that there are actually 502 components in the S&P500! We will *scrape* the website using the Python **requests** and **BeautifulSoup** libraries and then add the content directly to MySQL. Firstly make sure the libraries are installed:

```
pip install requests
pip install beautifulsoup4
```

The following code will use the requests and BeautifulSoup libraries to add the symbols directly to the MySQL database we created earlier. Remember to replace 'password' with your chosen password as created above:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# insert_symbols.py

from __future__ import print_function

import datetime
from math import ceil

import bs4
import MySQLdb as mdb
import requests


def obtain_parse_wiki_snp500():
    """
    Download and parse the Wikipedia list of S&P500
    constituents using requests and BeautifulSoup.

    Returns a list of tuples for to add to MySQL.
    """
    # Stores the current time, for the created_at record
    now = datetime.datetime.utcnow()

    # Use requests and BeautifulSoup to download the
    # list of S&P500 companies and obtain the symbol table
    response = requests.get(
        "http://en.wikipedia.org/wiki/List_of_S%26P_500_companies"
    )
    soup = bs4.BeautifulSoup(response.text)

    # This selects the first table, using CSS Selector syntax
    # and then ignores the header row ([1:])
    symbolslist = soup.select('table')[0].select('tr')[1:]

    # Obtain the symbol information for each
    # row in the S&P500 constituent table
    symbols = []
    for i, symbol in enumerate(symbolslist):
        tds = symbol.select('td')
        symbols.append(
            (
                tds[0].select('a')[0].text,  # Ticker
                'stock',
                tds[1].select('a')[0].text,  # Name
                tds[3].text,  # Sector
                'USD', now, now
            )
        )
    return symbols


def insert_snp500_symbols(symbols):
    """
    Insert the S&P500 symbols into the MySQL database.
    """
    # Connect to the MySQL instance
    db_host = 'localhost'
    db_user = 'sec_user'
    db_pass = 'password'
    db_name = 'securities_master'
    con = mdb.connect(
        host=db_host, user=db_user, passwd=db_pass, db=db_name
    )

    # Create the insert strings
    column_str = """ticker, instrument, name, sector,
                 currency, created_date, last_updated_date
                 """
    insert_str = ("%s, " * 7)[:-2]
    final_str = "INSERT INTO symbol (%s) VALUES (%s)" % \
        (column_str, insert_str)

    # Using the MySQL connection, carry out
    # an INSERT INTO for every symbol
    with con:
        cur = con.cursor()
        cur.executemany(final_str, symbols)


if __name__ == "__main__":
    symbols = obtain_parse_wiki_snp500()
    insert_snp500_symbols(symbols)
    print("%s symbols were successfully added." % len(symbols))
```

At this stage we'll have all 502 current symbol constituents of the S&P500 index in the database. Our next task is to actually obtain the historical pricing data from separate sources and match it up the symbols.

Price Retrieval

In order to obtain the historical data for the current S&P500 constituents, we must first query the database for the list of all the symbols.

Once the list of symbols, along with the symbol IDs, have been returned it is possible to call the Yahoo Finance API and download the historical pricing data for each symbol.

Once we have each symbol we can insert the data into the database in turn. Here's the Python code to carry this out:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# price_retrieval.py
```

```python
from __future__ import print_function

import datetime
import warnings

import MySQLdb as mdb
import requests


# Obtain a database connection to the MySQL instance
db_host = 'localhost'
db_user = 'sec_user'
db_pass = 'password'
db_name = 'securities_master'
con = mdb.connect(db_host, db_user, db_pass, db_name)


def obtain_list_of_db_tickers():
    """
    Obtains a list of the ticker symbols in the database.
    """
    with con:
        cur = con.cursor()
        cur.execute("SELECT id, ticker FROM symbol")
        data = cur.fetchall()
        return [(d[0], d[1]) for d in data]


def get_daily_historic_data_yahoo(
        ticker, start_date=(2000,1,1),
        end_date=datetime.date.today().timetuple()[0:3]
    ):
    """
    Obtains data from Yahoo Finance returns and a list of tuples.

    ticker: Yahoo Finance ticker symbol, e.g. "GOOG" for Google, Inc.
    start_date: Start date in (YYYY, M, D) format
    end_date: End date in (YYYY, M, D) format
    """
    # Construct the Yahoo URL with the correct integer query parameters
    # for start and end dates. Note that some parameters are zero-based!
    ticker_tup = (
        ticker, start_date[1]-1, start_date[2],
        start_date[0], end_date[1]-1, end_date[2],
        end_date[0]
    )
    yahoo_url = "http://ichart.finance.yahoo.com/table.csv"
    yahoo_url += "?s=%s&a=%s&b=%s&c=%s&d=%s&e=%s&f=%s"
    yahoo_url = yahoo_url % ticker_tup

    # Try connecting to Yahoo Finance and obtaining the data
    # On failure, print an error message.
    try:
        yf_data = requests.get(yahoo_url).text.split("\n")[1:-1]
        prices = []
        for y in yf_data:
            p = y.strip().split(',')
            prices.append(
                (datetime.datetime.strptime(p[0], '%Y-%m-%d'),
                p[1], p[2], p[3], p[4], p[5], p[6])
            )
    except Exception as e:
        print("Could not download Yahoo data: %s" % e)
    return prices


def insert_daily_data_into_db(
        data_vendor_id, symbol_id, daily_data
    ):
    """
    Takes a list of tuples of daily data and adds it to the
    MySQL database. Appends the vendor ID and symbol ID to the data.

    daily_data: List of tuples of the OHLC data (with
    adj_close and volume)
    """
    # Create the time now
    now = datetime.datetime.utcnow()

    # Amend the data to include the vendor ID and symbol ID
    daily_data = [
        (data_vendor_id, symbol_id, d[0], now, now,
        d[1], d[2], d[3], d[4], d[5], d[6])
        for d in daily_data
    ]

    # Create the insert strings
    column_str = """data_vendor_id, symbol_id, price_date, created_date,
                 last_updated_date, open_price, high_price, low_price,
                 close_price, volume, adj_close_price"""
    insert_str = ("%s, " * 11)[:-2]
    final_str = "INSERT INTO daily_price (%s) VALUES (%s)" % \
        (column_str, insert_str)

    # Using the MySQL connection, carry out an INSERT INTO for every symbol
    with con:
        cur = con.cursor()
        cur.executemany(final_str, daily_data)


if __name__ == "__main__":
    # This ignores the warnings regarding Data Truncation
    # from the Yahoo precision to Decimal(19,4) datatypes
    warnings.filterwarnings('ignore')

    # Loop over the tickers and insert the daily historical
    # data into the database
    tickers = obtain_list_of_db_tickers()
    lentickers = len(tickers)
    for i, t in enumerate(tickers):
        print(
            "Adding data for %s: %s out of %s" %
            (t[1], i+1, lentickers)
        )
        yf_data = get_daily_historic_data_yahoo(t[1])
        insert_daily_data_into_db('1', t[0], yf_data)
    print("Successfully added Yahoo Finance pricing data to DB.")
```

Note that there are certainly ways we can optimise this procedure. If we make use of the Python **ScraPy** library, for instance, we would gain high concurrency from the downloads, as ScraPy is built on the event-driven **Twisted** framework. At the moment each download will be carried out sequentially.

7.9 Retrieving Data from the Securities Master

Now that we've downloaded the historical pricing for all of the current S&P500 constituents we want to be able to access it within Python. The **pandas** library makes this extremely straightforward. Here's a script that obtains the Open-High-Low-Close (OHLC) data for the Google stock over a certain time period from our securities master database and outputs the *tail* of the dataset:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# retrieving_data.py

from __future__ import print_function

import pandas as pd
import MySQLdb as mdb


if __name__ == "__main__":
    # Connect to the MySQL instance
    db_host = 'localhost'
    db_user = 'sec_user'
    db_pass = 'password'
    db_name = 'securities_master'
    con = mdb.connect(db_host, db_user, db_pass, db_name)

    # Select all of the historic Google adjusted close data
    sql = """SELECT dp.price_date, dp.adj_close_price
             FROM symbol AS sym
             INNER JOIN daily_price AS dp
             ON dp.symbol_id = sym.id
             WHERE sym.ticker = 'GOOG'
             ORDER BY dp.price_date ASC;"""

    # Create a pandas dataframe from the SQL query
    goog = pd.read_sql_query(sql, con=con, index_col='price_date')

    # Output the dataframe tail
    print(goog.tail())
```

The output of the script follows:

```
            adj_close_price
price_date
2015-06-09           526.69
```

```
2015-06-10          536.69
2015-06-11          534.61
2015-06-12          532.33
2015-06-15          527.20
```

This is obviously only a simple script, but it shows how powerful having a locally-stored securities master can be. It is possible to backtest certain strategies extremely rapidly with this approach, as the input/output (I/O) speed from the database will be significantly faster than that of an internet connection.

Chapter 8

Processing Financial Data

In the previous chapter we outlined how to construct an equities-based securities master database. This chapter will discuss a topic that is not often considered to any great extent in the majority of trading books, that of processing financial market data prior to usage in a strategy test.

The discussion will begin with an overview of the different types of data that will be of interest to algorithmic traders. The frequency of the data will then be considered, from quarterly data (such as SEC reports) through to tick and order book data on the millisecond scale. Sources of such data (both free and commercial) will then be outlined along with code for obtaining the data. Finally, cleansing and preparation of the data for usage in strategies will be discussed.

8.1 Market and Instrument Classification

As algorithmic traders we are often interested in a broad range of financial markets data. This can range from underlying and derivative instrument time series prices, unstructured text-based data (such as news articles) through to corporate earnings information. This book will predominantly discuss financial time series data.

8.1.1 Markets

US and international equities, foreign exchange, commodities and fixed income are the primary sources of market data that will be of interest to an algorithmic trader. In the equities market it is still extremely common to purchase the underlying asset directly, while in the latter three markets highly liquid derivative instruments (futures, options or more exotic instruments) are more commonly used for trading purposes.

This broad categorisation essentially makes it relatively straightforward to deal in the equity markets, albeit with issues surrounding data handling of corporate actions (see below). Thus a large part of the retail algorithmic trading landscape will be based around equities, such as direct corporate shares or Exchange Traded Funds (ETFs). Foreign exchange ("forex") markets are also highly popular since brokers will allow *margin trading* on *percentage in point* (PIP) movements. A *pip* is one unit of the fourth decimal point in a currency rate. For currencies denominated in US dollars this is equivalent to 1/100th of a cent.

Commodities and fixed income markets are harder to trade in the underlying directly. A retail algorithmic trader is often not interested in delivering barrels of oil to an oil depot! Instead, futures contracts on the underlying asset are used for speculative purposes. Once again, margin trading is employed allowing extensive leverage on such contracts.

8.1.2 Instruments

A wide range of underlying and derivative instruments are available to the algorithmic trader. The following table describes the common use cases of interest.

Market	Instruments
Equities/Indices	Stock, ETFs, Futures, Options
Foreign Exchange	Margin/Spot, ETFs, Futures, Options
Commodities	Futures, Options
Fixed Income	Futures, Options

For the purposes of this book we will concentrate almost exclusively upon equities and ETFs to simplify the implementation.

8.1.3 Fundamental Data

Although algorithmic traders primarily carry out analysis of financial price time series, fundamental data (of varying frequencies) is also often added to the analysis. So-called *Quantitative Value* (QV) strategies rely heavily on the accumulation and analysis of fundamental data, such as macroeconomic information, corporate earnings histories, inflation indexes, payroll reports, interest rates and SEC filings. Such data is often also in temporal format, albeit on much larger timescales over months, quarters or years. QV strategies also operate on these timeframes.

This book will not discuss QV strategies or large time-scale fundamental driven strategies to any great extent, rather will concentrate on the daily or more frequent strategies derived mostly from price action.

8.1.4 Unstructured Data

Unstructured data consists of *documents* such as news articles, blog posts, papers or reports. Analysis of such data can be complicated as it relies on *Natural Language Processing* (NLP) techniques. One such use of analysing unstructured data is in trying to determine the *sentiment* context. This can be useful in driving a trading strategy. For instance, by classifying texts as "bullish", "bearish" or "neutral" a set of trading signals could be generated. The term for this process is *sentiment analysis.*

Python provides an extremely comprehensive library for the analysis of text data known as the Natural Language Toolkit (NLTK). Indeed an O'Reilly book on NLTK can be downloaded for free via the authors' website - Natural Language Processing with Python[3].

Full-Text Data

There are numerous sources of full-text data that may be useful for generating a trading strategy. Popular financial sources such as Bloomberg and the Financial Times, as well as financial commentary blogs such as Seeking Alpha and ZeroHedge, provide significant sources of text to analyse. In addition, proprietary news feeds as provided by data vendors are also good sources of such data.

In order to obtain data on a larger scale, one must make use of "web scraping" tools, which are designed to automate the downloading of websites en-masse. Be careful here as automated web-scraping tools sometimes breach the Terms Of Service for these sites. Make sure to check before you begin downloading this sort of data. A particularly useful tool for web scraping, which makes the process efficient and structured, is the ScraPy library.

Social Media Data

In the last few years there has been significant interest in obtaining sentiment information from social media data, particularly via the Twitter micro-blogging service. Back in 2011, a hedge fund was launched around Twitter sentiment, known as Derwent Capital. Indeed, academic studies[4] have shown evidence that it is possible to generate a degree of predictive capability based on such sentiment analysis.

While sentiment analysis is out of the scope of this book if you wish to carry out research into sentiment, then there are two books[15, 14] by Matt Russell on obtaining social media data via the public APIs provided by these web services.

8.2 Frequency of Data

Frequency of data is one of the most important considerations when designing an algorithmic trading system. It will impact every design decision regarding the storage of data, backtesting a strategy and executing an algorithm.

Higher frequency strategies are likely to lead to more statistically robust analysis, simply due to the greater number of data points (and thus trades) that will be used. HFT strategies often require a significant investment of time and capital for development of the necessary software to carry them out.

Lower frequency strategies are easier to develop and deploy, since they require less automation. However, they will often generate far less trades than a higher-frequency strategy leading to a less statistically robust analysis.

8.2.1 Weekly and Monthly Data

Fundamental data is often reported on a weekly, monthly, quarterly or even yearly basis. Such data include payroll data, hedge fund performance reports, SEC filings, inflation-based indices (such as the Consumer Price Index, CPI), economic growth and corporate accounts.

Storage of such data is often suited to unstructured databases, such as MongoDB, which can handle hierarchically-nested data and thus allowing a reasonable degree of querying capability. The alternative is to store flat-file text in a RDBMS, which is less appropriate, since full-text querying is trickier.

8.2.2 Daily Data

The majority of retail algorithmic traders make use of daily ("end of day"/EOD) financial time series data, particularly in equities and foreign exchange. Such data is freely available (see below), but often of questionable quality and subject to certain biases. End-of-day data is often stored in RDBMS, since the nature of ticker/symbol mapping naturally applies to the relational model.

EOD data does not entail particularly large storage requirements. There are 252 trading days in a year for US exchanges and thus for a decade there will be 2,520 bars per security. Even with a universe of 10,000 symbols this is 25,200,000 bars, which can easily be handled within a relational database environment.

8.2.3 Intraday Bars

Intraday strategies often make use of hourly, fifteen-, five-, one-minutely or secondly OHLCV bars. Intraday feed providers such as QuantQuote and DTN IQFeed will often provide minutely or secondly bars based on their tick data.

Data at such frequencies will possess many "missing" bars simply because no trades were carried out in that time period. Pandas can be used to pad these values forward, albeit with a decrease in data accuracy. In addition pandas can also be used to create data on less granular timescales if necessary.

For a ten year period, minutely data will generate almost one million bars per security. Similarly for secondly data the number of data points over the same period will total almost sixty million per security. Thus to store one thousand of such securities will lead to sixty billion bars of data. This is a large amount of data to be kept in an RDBMS and consequently more sophisticated approaches are required.

Storage and retrieval of secondly data on this magnitude is somewhat outside the scope of this book so I won't discuss it further.

8.2.4 Tick and Order Book Data

When a trade is filled at an exchange, or other venue, a *tick* is generated. Tick feeds consist of all such transactions *per exchange*. Retail tick feeds are stored with each datum having a timestamp accurate to the millisecond level. Tick data often also includes the updated best bid/ask price. The storage of tick data is well beyond the scope of this book but needless to say the volumes of such data are substantial. Common storage mechanisms include HDF5, kdb and simply flat-file/CSV.

Multiple *limit orders* at an exchange lead to the concept of an *order book*. This is essentially the list of all bid and ask limit orders at certain volumes for each market participant. It leads to the definition of the *bid-ask spread* (or simply the "spread"), which is the smallest difference in the bid and ask prices for the "top of book" orders. Creating a historical representation, or a market simulator, of a limit order book is usually necessary for carrying out ultra high frequency trading (UHFT) strategies. The storage of such data is complex and as such will be outside the scope of this book.

8.3 Sources of Data

There are numerous sources and vendors of financial data. They vary substantially in breadth, timeliness, quality and price.

Broadly speaking, financial market data provided on a delayed daily frequency or longer is available freely, albeit with dubious overall quality and the potential for survivorship bias. To obtain intraday data it is usually necessary to purchase a commercial data feed. The vendors of such feeds vary tremendously in their customer service capability, overall feed quality and breadth of instruments.

8.3.1 Free Sources

Free end-of-day bar data, which consists of Open-High-Low-Close-Volume (OHLCV) prices for instruments, is available for a wide range of US and international equities and futures from Yahoo Finance, Google Finance and Quandl.

Yahoo Finance

Yahoo Finance is the "go to" resource when forming an end-of-day US equities database. The breadth of data is extremely comprehensive, listing thousands of traded equities. In addition stock-splits and dividends are handled using a back-adjustment method, arising as the "Adj Close" column in the CSV output from the API (which we discuss below). Thus the data allows algorithmic traders to get started rapidly and for zero cost.

I have personally had a lot of experience in cleaning Yahoo data. I have to remark that the data can be quite erroneous. Firstly, it is subject to a problem known as *backfilling*. This problem occurs when past historical data is corrected at a future date, leading to poor quality backtests that change as your own database is re-updated. To handle this problem, a logging record is usually added to the securities master (in an appropriate logging table) whenever a historical data point is modified.

Secondly, the Yahoo feed only aggregates prices from a few sources to form the OHLCV points. This means that values around the open, high, low and close can be deceptive, as other exchanges/liquidity sources may have executed differing prices in excess of the values.

Thirdly, I have noticed that when obtaining financial data *en-masse* from Yahoo, that errors do creep into the API. For instance, multiple calls to the API with identical date/ticker parameters occasionally lead to differing result sets. This is clearly a substantial problem and must be carefully checked for.

In summary be prepared to carry out some extensive data cleansing on Yahoo Finance data, if you choose to use it to populate a large securities master, and need highly accurate data.

Quandl

Quandl is a relatively new service which purports to be *"The easiest way to find and use numerical data on the web"*. I believe they are well on the way to achieving that goal! The service provides a substantial daily data set of US and international equities, interest rates, commodities/futures, foreign exchange and other economic data. In addition, the database is continually expanded and the project is very actively maintained.

All of the data can be accessed by a very modern HTTP API (CSV, JSON, XML or HTML), with plugins for a wide variety of programming languages including R, Python, Matlab, Excel, Stata, Maple, C#, EViews, Java, C/C++, .NET, Clojure and Julia. Without an account 50 calls to the API are allowed per day, but this can be increased to 500 if registering an account. In fact, calls can be updated to 5,000 per hour if so desired by contacting the team.

I have not had a great deal of experience with Quandl "at scale" and so I can't comment on the level of errors within the dataset, but my feeling is that any errors are likely to be constantly reported and corrected. Thus they are worth considering as a primary data source for an end-of-day securities master.

Later in the chapter we will discuss how to obtain US Commodities Futures data from Quandl with Python and pandas.

8.3.2 Commercial Sources

In order to carry out intraday algorithmic trading it is usually necessary to purchase a commercial feed. Pricing can range anywhere from $30 per month to around $500 per month for "retail level" feeds. Institutional quality feeds will often be in the low-to-mid four figure range per month and as such I won't discuss them here.

EODData

I have utilised EODData in a fund context, albeit only with daily data and predominantly for foreign exchange. Despite their name they do provide a degree of intraday sources. The cost is $25 per month for their "platinum" package.

The resource is very useful for finding a full list of traded symbols on global exchanges, but remember that this will be subject to survivorship bias as I believe the list represents current listed entities.

Unfortunately (at least back in 2010) I found that the stock split feed was somewhat inaccurate (at least when compared to Morningstar information). This lead to some substantial spike issues (see below) in the data, which increased friction in the data cleansing process.

DTN IQFeed

DTN IQFeed are one of the most popular data feeds for the high-end retail algorithmic trader. They claim to have over 80,000 customers. They provide real-time tick-by-tick data unfiltered from the exchange as well as a large quantity of historic data.

The pricing starts at $50 per month, but in reality will be in the $150-$200 per month range once particular services are selected and exchange fees are factored in. I utilise DTN IQFeed for all of my intraday equities and futures strategies. In terms of historical data, IQFeed provide for equities, futures and options:

- 180 calendar days of tick (every trade)
- 7+ years of 1 minute historical bars
- 15+ years of daily historical bars

The major disadvantage is that the DTN IQFeed software (the mini-server, not the charting tools) will only run on Windows. This may not be a problem if all of your algorithmic trading is carried out in this operating system, but I personally develop all my strategies in Ubuntu Linux. However, although I have not actively tested it, I have heard it is possible to run DTN IQFeed under the WINE emulator.

Below we will discuss how to obtain data from IQFeed using Python in Windows.

QuantQuote

QuantQuote provide reasonably priced historical minute-, second- and tick-level data for US equities going back to 1998. In addition they provide institutional level real-time tick feeds, although this is of less interest to retail algorithmic traders. One of the main benefits of QuantQuote is that their data is provided free of survivorship bias, due to their TickMap symbol-matching software and inclusion of all stocks within a certain index through time.

As an example, to purchase the entire history of the S&P500 going back to 1998 in minutely-bars, inclusive of de-listed stocks, the cost at the time of writing was $895. The pricing scales with increasing frequency of data.

QuantQuote is currently the primary provider of market data to the QuantConnect web-based backtesting service. QuantQuote go to great lengths to ensure minimisation of error, so if you are looking for a US equities only feed at high resolution, then you should consider using their service.

8.4 Obtaining Data

In this section we are going to discuss how to use Quandl, pandas and DTN IQFeed to download financial market data across a range of markets and timeframes.

8.4.1 Yahoo Finance and Pandas

The pandas library makes it exceedingly simple to download EOD data from Yahoo Finance. Pandas ships with a DataReader component that ties into Yahoo Finance (among other sources). Specifying a symbol with a start and end date is sufficient to download an EOD series into a pandas DataFrame, which allows rapid vectorised operations to be carried out:

```python
from __future__ import print_function

import datetime
import pandas.io.data as web

if __name__ == "__main__":
    spy = web.DataReader(
        "SPY", "yahoo",
        datetime.datetime(2007,1,1),
        datetime.datetime(2015,6,15)
    )
    print(spy.tail())
```

The output is given below:

```
                  Open        High         Low       Close     Volume  \
Date
2015-06-09  208.449997  209.100006  207.690002  208.449997   98148200
2015-06-10  209.369995  211.410004  209.300003  210.960007  129936200
2015-06-11  211.479996  212.089996  211.199997  211.649994   72672100
2015-06-12  210.639999  211.479996  209.679993  209.929993  127811900
2015-06-15  208.639999  209.449997  207.789993  209.100006  121425800

             Adj Close
Date
2015-06-09  208.449997
2015-06-10  210.960007
2015-06-11  211.649994
2015-06-12  209.929993
2015-06-15  209.100006
```

Note that in pandas 0.17.0, `pandas.io.data` *will be replaced by a separate* `pandas-datareader` *package. However, for the time being (i.e. pandas versions 0.16.x) the syntax to import the data reader is* `import pandas.io.data as web`.

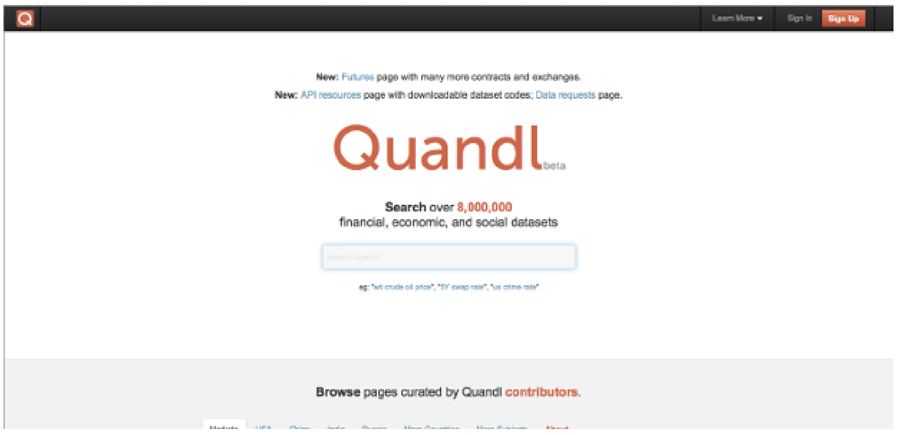
In the next section we will use Quandl to create a more comprehensive, permanent download solution.

8.4.2 Quandl and Pandas

Up until recently it was rather difficult and expensive to obtain consistent futures data across exchanges in frequently updated manner. However, the release of the Quandl service has changed the situation dramatically, with financial data in some cases going back to the 1950s. In this section we will use Quandl to download a set of end-of-day futures contracts across multiple delivery dates.

Signing Up For Quandl

The first thing to do is sign up to Quandl. This will increase the daily allowance of calls to their API. Sign-up grants 500 calls per day, rather than the default 50. Visit the site at www.quandl.com:



New: Futures page with many more contracts and exchanges.
New: API resources page with downloadable dataset codes; Data requests page.

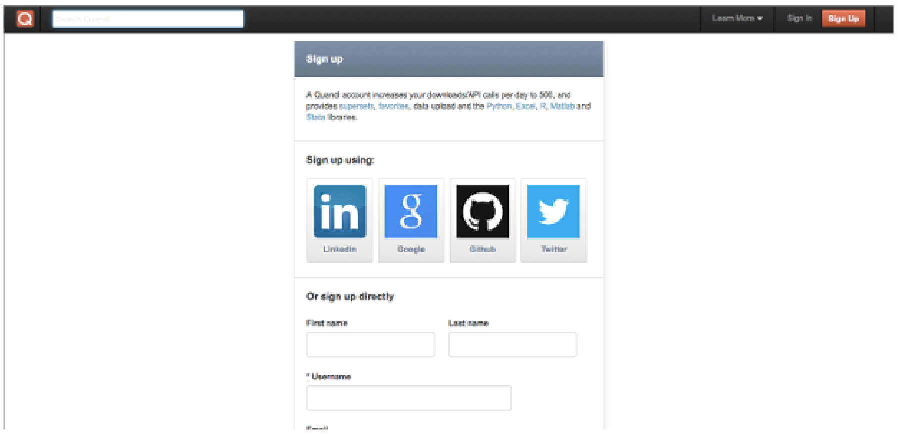
Search over 8,000,000
financial, economic, and social datasets

eg: "wti crude oil price", "5Y swap rate", "us crime rate"

Browse pages curated by Quandl contributors.

Figure 8.1: The Quandl homepage

Click on the sign-up button on the top right:



Sign up

A Quandl account increases your downloads/API calls per day to 500, and provides supersets, favorites, data upload and the Python, Excel, R, Matlab and Stata libraries.

Sign up using:

Or sign up directly

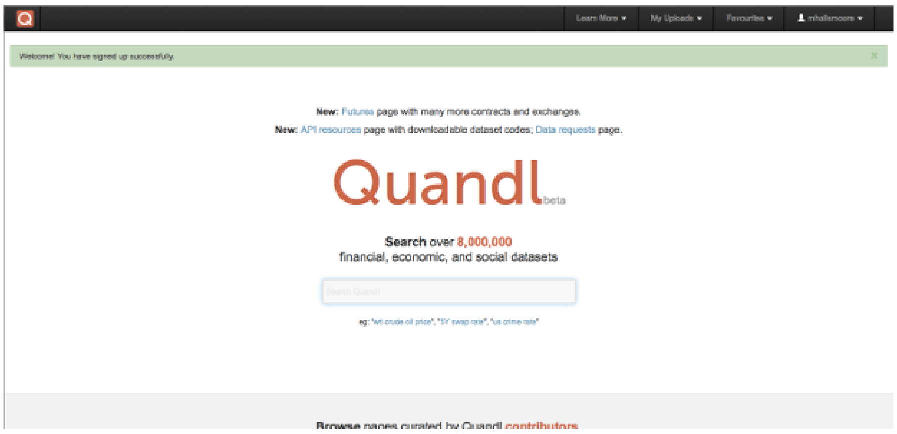
First name

Last name

* Username

Figure 8.2: The Quandl sign-up page

Once you're signed in you'll be returned to the home page:



Welcome! You have signed up successfully.

New: Futures page with many more contracts and exchanges.
New: API resources page with downloadable dataset codes; Data requests page.

Quandl beta

Search over 8,000,000
financial, economic, and social datasets

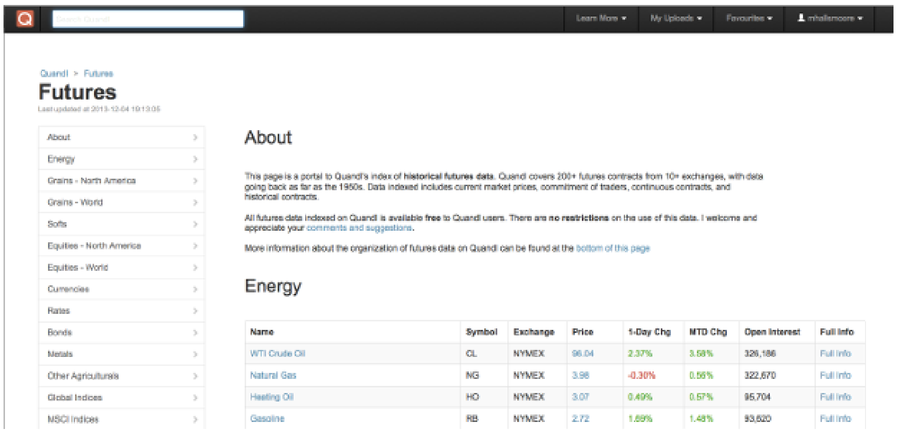
eg: "wti crude oil price", "5Y swap rate", "us crime rate"

Browse pages curated by Quandl contributors

Figure 8.3: The Quandl authorised home page

Quandl Futures Data

Now click on the "New: Futures page..." link to get to the futures homepage:



Futures
Last updated at 2013-12-04 10:13:05

About

This page is a portal to Quandl's index of **historical futures data**. Quandl covers 200+ futures contracts from 10+ exchanges, with data going back as far as the 1950s. Data indexed includes current market prices, commitment of traders, continuous contracts, and historical contracts.

All futures data indexed on Quandl is available **free** to Quandl users. There are **no restrictions** on the use of this data. I welcome and appreciate your comments and suggestions.

More information about the organization of futures data on Quandl can be found at the bottom of this page

Energy

Name	Symbol	Exchange	Price	1-Day Chg	MTD Chg	Open Interest	Full Info
WTI Crude Oil	CL	NYMEX	96.04	2.37%	3.58%	326,186	Full Info
Natural Gas	NG	NYMEX	3.98	-0.30%	0.56%	322,670	Full Info
Heating Oil	HO	NYMEX	3.07	0.49%	0.57%	95,704	Full Info
Gasoline	RB	NYMEX	2.72	1.69%	1.48%	93,620	Full Info

Figure 8.4: The Quandl futures contracts home page

For this tutorial we will be considering the highly liquid E-Mini S&P500 futures contract, which has the futures symbol ES. To download other contracts the remainder of this tutorial can be carried out with additional symbols replacing the reference to ES.

Click on the E-Mini S&P500 link (or your chosen futures symbol) and you'll be taken to the following screen:

Scrolling further down the screen displays the list of historical contracts going back to 1997:

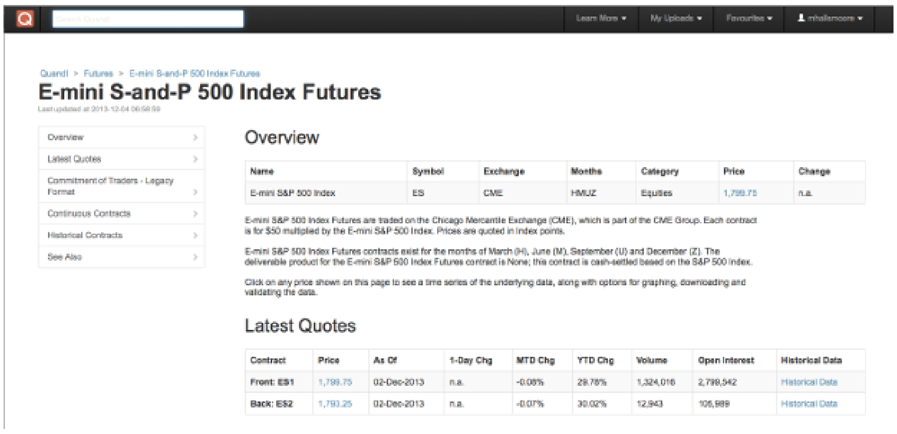
Click on one of the individual contracts. As an example, I have chosen ESZ2014, which refers to the contract for December 2014 'delivery'. This will display a chart of the data:

By clicking on the "Download" button the data can be obtained in multiple formats: HTML, CSV, JSON or XML. In addition we can download the data directly into a pandas DataFrame using the Python bindings. While the latter is useful for quick "prototyping" and exploration of the data, in this section we are considering the development of a longer term data store. Click the download button, select "CSV" and then copy and paste the API call:

The API call will have the following form:

```
http://www.quandl.com/api/v1/datasets/OFDP/FUTURE_ESZ2014.csv?
&auth_token=MY_AUTH_TOKEN&trim_start=2013-09-18
&trim_end=2013-12-04&sort_order=desc
```

The authorisation token has been redacted and replaced with MY_AUTH_TOKEN. It will be necessary to copy the alphanumeric string between "auth_token=" and "&trim_start" for



E-mini S-and-P 500 Index Futures

Last updated at 2013-12-04 06:58:59

Overview

Name	Symbol	Exchange	Months	Category	Price	Change
E-mini S&P 500 Index	ES	CME	HMUZ	Equities	1,799.75	n.a.

E-mini S&P 500 Index Futures are traded on the Chicago Mercantile Exchange (CME), which is part of the CME Group. Each contract is for $50 multiplied by the E-mini S&P 500 Index. Prices are quoted in index points.

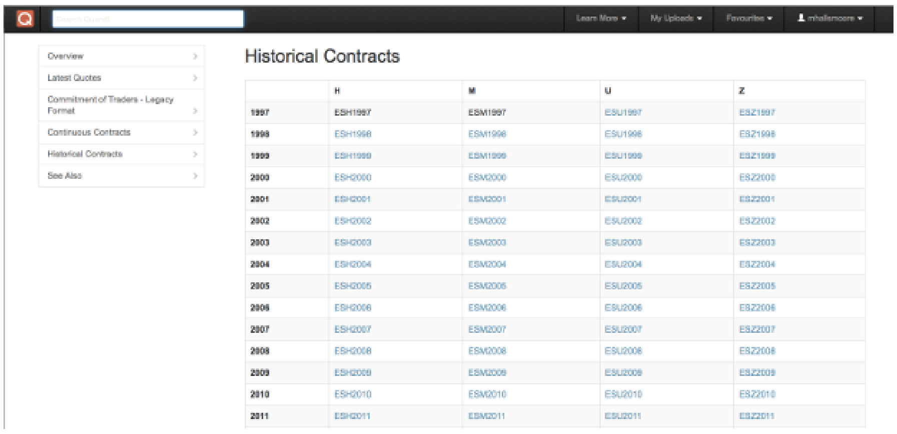
E-mini S&P 500 Index Futures contracts exist for the months of March (H), June (M), September (U) and December (Z). The deliverable product for the E-mini S&P 500 Index Futures contract is None; this contract is cash-settled based on the S&P 500 Index.

Click on any price shown on this page to see a time series of the underlying data, along with options for graphing, downloading and validating the data.

Latest Quotes

Contract	Price	As Of	1-Day Chg	MTD Chg	YTD Chg	Volume	Open Interest	Historical Data
Front: ES1	1,799.75	02-Dec-2013	n.a.	-0.08%	29.78%	1,324,016	2,798,542	Historical Data
Back: ES2	1,793.25	02-Dec-2013	n.a.	-0.07%	30.02%	12,943	105,989	Historical Data

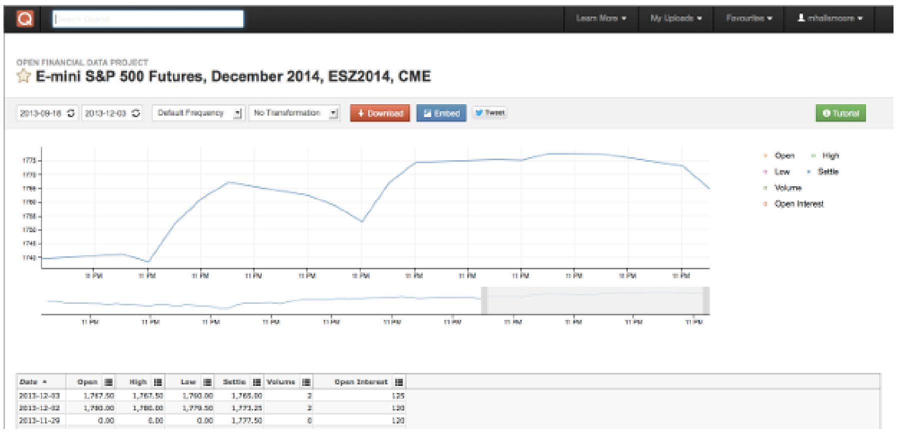
Figure 8.5: E-Mini S&P500 contract page



Historical Contracts

	H	M	U	Z
1997	ESH1997	ESM1997	ESU1997	ESZ1997
1998	ESH1998	ESM1998	ESU1998	ESZ1998
1999	ESH1999	ESM1999	ESU1999	ESZ1999
2000	ESH2000	ESM2000	ESU2000	ESZ2000
2001	ESH2001	ESM2001	ESU2001	ESZ2001
2002	ESH2002	ESM2002	ESU2002	ESZ2002
2003	ESH2003	ESM2003	ESU2003	ESZ2003
2004	ESH2004	ESM2004	ESU2004	ESZ2004
2005	ESH2005	ESM2005	ESU2005	ESZ2005
2006	ESH2006	ESM2006	ESU2006	ESZ2006
2007	ESH2007	ESM2007	ESU2007	ESZ2007
2008	ESH2008	ESM2008	ESU2008	ESZ2008
2009	ESH2009	ESM2009	ESU2009	ESZ2009
2010	ESH2010	ESM2010	ESU2010	ESZ2010
2011	ESH2011	ESM2011	ESU2011	ESZ2011

Figure 8.6: E-Mini S&P500 historical contracts



OPEN FINANCIAL DATA PROJECT

E-mini S&P 500 Futures, December 2014, ESZ2014, CME

Date	Open	High	Low	Settle	Volume	Open Interest
2013-12-03	1,767.50	1,767.50	1,760.00	1,765.00	2	125
2013-12-02	1,780.00	1,780.00	1,779.50	1,773.25	2	120
2013-11-29	0.00	0.00	0.00	1,777.50	0	120

Figure 8.7: Chart of ESZ2014 (December 2014 delivery)

later usage in the Python script below. Do not share it with anyone as it is your unique authorisation token for Quandl downloads and is used to determine your download rate for the day.

This API call will form the basis of an automated script which we will write below to download a subset of the entire historical futures contract.

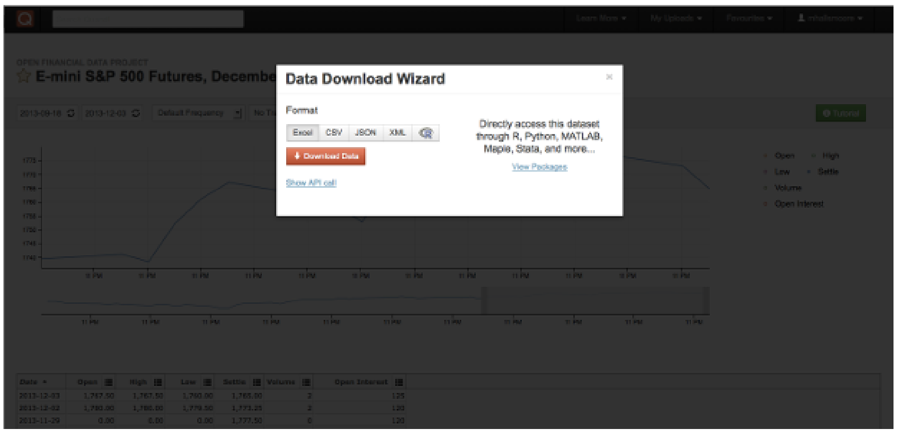

E-mini S&P 500 Futures, Decembe
Data Download Wizard
Format
Excel
CSV
JSON
XML
Download Data
Show API call
Directly access this dataset through R, Python, MATLAB, Maple, Stata, and more...
View Packages

Figure 8.8: Download modal for ESZ2014 CSV file

Downloading Quandl Futures into Python

Because we are interested in using the futures data long-term as part of a wider securities master database strategy we want to store the futures data to disk. Thus we need to create a directory to hold our E-Mini contract CSV files. In Mac/Linux (within the terminal/console) this is achieved by the following command:

```
cd /PATH/TO/YOUR/quandl_data.py
mkdir -p quandl/futures/ES
```

Note: Replace `/PATH/TO/YOUR` above with the directory where your `quandl_data.py` file is located.

This creates a subdirectory of called `quandl`, which contains two further subdirectories for futures and for the ES contracts in particular. This will help us to organise our downloads in an ongoing fashion.

In order to carry out the download using Python we will need to import some libraries. In particular we will need **requests** for the download and **pandas** and **matplotlib** for plotting and data manipulation:

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

# quandl_data.py

from __future__ import print_function

import matplotlib.pyplot as plt
import pandas as pd
import requests
```

The first function within the code will generate the list of futures symbols we wish to download. I've added keyword parameters for the start and end years, setting them to reasonable values of 2010 and 2014. You can, of course, choose to use other timeframes:

```python
def construct_futures_symbols(
        symbol, start_year=2010, end_year=2014
    ):
    """
    Constructs a list of futures contract codes
    for a particular symbol and timeframe.
    """
    futures = []
    # March, June, September and
    # December delivery codes
    months = 'HMUZ'
    for y in range(start_year, end_year+1):
        for m in months:
            futures.append("%s%s%s" % (symbol, m, y))
    return futures
```

Now we need to loop through each symbol, obtain the CSV file from Quandl for that particular contract and subsequently write it to disk so we can access it later:

```
def download_contract_from_quandl(contract, dl_dir):
    """
    Download an individual futures contract from Quandl and then
    store it to disk in the 'dl_dir' directory. An auth_token is
    required, which is obtained from the Quandl upon sign-up.
    """
    # Construct the API call from the contract and auth_token
    api_call = "http://www.quandl.com/api/v1/datasets/"
    api_call += "OFDP/FUTURE_%s.csv" % contract
    # If you wish to add an auth token for more downloads, simply
    # comment the following line and replace MY_AUTH_TOKEN with
    # your auth token in the line below
    params = "?sort_order=asc"
    #params = "?auth_token=MY_AUTH_TOKEN&sort_order=asc"
    full_url = "%s%s" % (api_call, params)

    # Download the data from Quandl
    data = requests.get(full_url).text

    # Store the data to disk
    fc = open('%s/%s.csv' % (dl_dir, contract), 'w')
    fc.write(data)
    fc.close()
```

Now we tie the above two functions together to download all of the desired contracts:

```
def download_historical_contracts(
        symbol, dl_dir, start_year=2010, end_year=2014
    ):
    """
    Downloads all futures contracts for a specified symbol
    between a start_year and an end_year.
    """
    contracts = construct_futures_symbols(
        symbol, start_year, end_year
    )
    for c in contracts:
        print("Downloading contract: %s" % c)
        download_contract_from_quandl(c, dl_dir)
```

Finally, we can add one of the futures prices to a pandas dataframe using the main function. We can then use matplotlib to plot the settle price:

```
if __name__ == "__main__":
    symbol = 'ES'

    # Make sure you've created this
    # relative directory beforehand
    dl_dir = 'quandl/futures/ES'

    # Create the start and end years
    start_year = 2010
    end_year = 2014

    # Download the contracts into the directory
    download_historical_contracts(
        symbol, dl_dir, start_year, end_year
    )

    # Open up a single contract via read_csv
    # and plot the settle price
    es = pd.io.parsers.read_csv(
        "%s/ESH2010.csv" % dl_dir, index_col="Date"
    )
    es["Settle"].plot()
    plt.show()
```

The output of the plot is given in Figure 8.4.2.

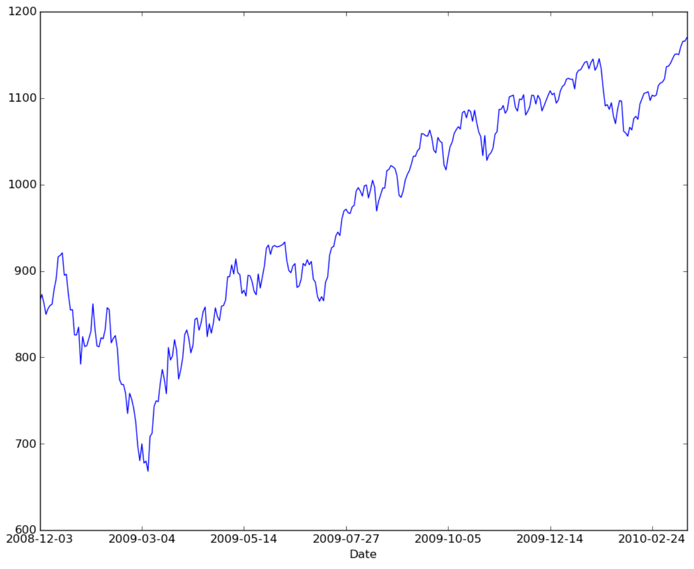


Figure 8.9: ESH2010 Settle Price

The above code can be modified to collect any combination of futures contracts from Quandl as necessary. Remember that unless a higher API request is made, the code will be limited to making 50 API requests per day.

8.4.3 DTN IQFeed

For those of you who possess a DTN IQFeed subscription, the service provides a client-server mechanism for obtaining intraday data. For this to work it is necessary to download the IQLink server and run it on Windows. Unfortunately, it is tricky to execute this server on Mac or Linux unless making use of the WINE emulator. However once the server is running it can be connected to via a socket at which point it can be queried for data.

In this section we will obtain minutely bar data for a pair of US ETFs from January 1st 2007 onwards using a Python socket interface. Since there are approximately 252 trading days within each year for US markets, and each trading day has 6.5 hours of trading, this will equate to at least 650,000 bars of data, each with seven data points: Timestamp, Open, Low, High, Close, Volume and Open Interest.

I have chosen the SPY and IWM ETFs to download to CSV. Make such to start the IQLink program in Windows before executing this script:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# iqfeed.py

import sys
import socket

def read_historical_data_socket(sock, recv_buffer=4096):
    """
    Read the information from the socket, in a buffered
    fashion, receiving only 4096 bytes at a time.

    Parameters:
    sock - The socket object
    recv_buffer - Amount in bytes to receive per read
    """
    buffer = ""
    data = ""
    while True:
        data = sock.recv(recv_buffer)
        buffer += data

        # Check if the end message string arrives
        if "!ENDMSG!" in buffer:
            break

    # Remove the end message string
    buffer = buffer[:-12]
    return buffer

if __name__ == "__main__":
    # Define server host, port and symbols to download
    host = "127.0.0.1"  # Localhost
    port = 9100  # Historical data socket port
    syms = ["SPY", "IWM"]

    # Download each symbol to disk
    for sym in syms:
        print "Downloading symbol: %s..." % sym

        # Construct the message needed by IQFeed to retrieve data
        message = "HIT,%s,60,20070101 075000,,,093000,160000,1\n" % sym

        # Open a streaming socket to the IQFeed server locally
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect((host, port))

        # Send the historical data request
        # message and buffer the data
        sock.sendall(message)
        data = read_historical_data_socket(sock)
        sock.close

        # Remove all the endlines and line-ending
        # comma delimiter from each record
        data = "".join(data.split("\r"))
        data = data.replace(",\n","\n")[:-1]

        # Write the data stream to disk
        f = open("%s.csv" % sym, "w")
        f.write(data)
        f.close()
```

With additional subscription options in the DTN IQFeed account, it is possible to download individual futures contracts (and back-adjusted continuous contracts), options and indices. DTN IQFeed also provides real-time tick streaming, but this form of data falls outside the scope of the book.

8.5 Cleaning Financial Data

Subsequent to the delivery of financial data from vendors it is necessary to perform *data cleansing*. Unfortunately this can be a painstaking process, but a hugely necessary one. There are multiple issues that require resolution: Incorrect data, consideration of data aggregation and backfilling. Equities and futures contracts possess their own unique challenges that must be dealt with prior to strategy research, including back/forward adjustment, continuous contract stitching and corporate action handling.

8.5.1 Data Quality

The reputation of a data vendor will often rest on its (perceived) data quality. In simple terms, bad or missing data leads to erroneous trading signals and thus potential loss. Despite this fact, many vendors are still plagued with poor or inconsistent data quality. Thus there is always a cleansing process necessary to be carried.

The main culprits in poor data quality are conflicting/incorrect data, opaque aggregation of multiple data sources and error correction ("backfilling").

Conflicting and Incorrect Data

Bad data can happen anywhere in the stream. Bugs in the software at an exchange can lead to erroneous prices when matching trades. This filters through to the vendor and subsequently the trader. Reputable vendors will attempt to flag upstream "bad ticks" and will often leave the "correction" of these points to the trader.

8.5.2 Continuous Futures Contracts

In this section we are going to discuss the characteristics of futures contracts that present a data challenge from a backtesting point of view. In particular, the notion of the "continuous contract". We will outline the main difficulties of futures and provide an implementation in Python with pandas that can partially alleviate the problems.

Brief Overview of Futures Contracts

Futures are a form of *contract* drawn up between two parties for the purchase or sale of a quantity of an underlying asset at a specified date in the future. This date is known as the *delivery* or *expiration*. When this date is reached the buyer must deliver the physical underlying (or cash equivalent) to the seller for the price agreed at the contract formation date.

In practice futures are traded on exchanges (as opposed to *Over The Counter* - OTC trading) for standardised quantities and qualities of the underlying. The prices are *marked to market* every day. Futures are incredibly liquid and are used heavily for speculative purposes. While futures were often utilised to hedge the prices of agricultural or industrial goods, a futures contract can be formed on any tangible or intangible underlying such as stock indices, interest rates of foreign exchange values.

A detailed list of all the symbol codes used for futures contracts across various exchanges can be found on the CSI Data site: Futures Factsheet.

The main difference between a futures contract and equity ownership is the fact that a futures contract has a limited window of availability by virtue of the expiration date. At any one instant there will be a variety of futures contracts on the same underlying all with varying dates of expiry. The contract with the nearest date of expiry is known as the *near contract*. The problem we face as quantitative traders is that at any point in time we have a choice of multiple contracts with which to trade. Thus we are dealing with an overlapping set of time series rather than a continuous stream as in the case of equities or foreign exchange.

The goal of this section is to outline various approaches to constructing a continuous stream of contracts from this set of multiple series and to highlight the tradeoffs associated with each technique.

Forming a Continuous Futures Contract

The main difficulty with trying to generate a continuous contract from the underlying contracts with varying deliveries is that the contracts do not often trade at the same prices. Thus situations arise where they do not provide a smooth splice from one to the next. This is due to contango and backwardation effects. There are various approaches to tackling this problem, which we now discuss.

Unfortunately there is no single "standard" method for joining futures contracts together in the financial industry. Ultimately the method chosen will depend heavily upon the strategy employing the contracts and the method of execution. Despite the fact that no single method exists there are some common approaches:

The **Back/Forward ("Panama") Adjustment** method alleviates the "gap" across multiple contracts by shifting each contract such that the individual deliveries join in a smooth manner to the adjacent contracts. Thus the open/close across the prior contracts at expiry matches up.

The key problem with the Panama method includes the introduction of a trend bias, which will introduce a large drift to the prices. This can lead to negative data for sufficiently historical contracts. In addition there is a loss of the *relative* price differences due to an absolute shift in values. This means that returns are complicated to calculate (or just plain incorrect).

The **Proportionality Adjustment** approach is similar to the adjustment methodology of handling stock splits in equities. Rather than taking an absolute shift in the successive contracts, the ratio of the older settle (close) price to the newer open price is used to proportionally adjust the prices of historical contracts. This allows a continous stream without an interruption of the calculation of percentage returns.

The main issue with proportional adjustment is that any trading strategies reliant on an absolute price level will also have to be similarly adjusted in order to execute the correct signal. This is a problematic and error-prone process. Thus this type of continuous stream is often only useful for summary statistical analysis, as opposed to direct backtesting research.

The **Rollover/Perpetual Series** method creates a continuous contract of successive contracts by taking a linearly weighted proportion of each contract over a number of days to ensure a smoother transition between each.

For example consider five smoothing days. The price on day 1, P_1, is equal to 80% of the far contract price (F_1) and 20% of the near contract price (N_1). Similarly, on day 2 the price is $P_2 = 0.6 \times F_2 + 0.4 \times N_2$. By day 5 we have $P_5 = 0.0 \times F_5 + 1.0 \times N_5 = N_5$ and the contract then just becomes a continuation of the near price. Thus after five days the contract is smoothly transitioned from the far to the near.

The problem with the rollover method is that it requires trading on all five days, which can increase transaction costs. There are other less common approaches to the problem but we will avoid them here.

The remainder of the section will concentrate on implementing the perpetual series method as this is most appropriate for backtesting. It is a useful way to carry out *strategy pipeline research*.

We are going to stitch together the WTI Crude Oil "near" and "far" futures contract (symbol CL) in order to generate a continuous price series. At the time of writing (January 2014), the near contract is CLF2014 (January) and the far contract is CLG2014 (February).

In order to carry out the download of futures data I've made use of the Quandl plugin. Make sure to set the correct Python virtual environment on your system and install the Quandl package by typing the following into the terminal:

```
pip install Quandl
```

Now that the Quandl package is intalled, we need to make use of NumPy and pandas in order to carry out the rollover construction. Create a new file and enter the following import statements:

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

# cont_futures.py

from __future__ import print_function

import datetime

import numpy as np
import pandas as pd
import Quandl
```

The main work is carried out in the `futures_rollover_weights` function. It requires a starting date (the first date of the near contract), a dictionary of contract settlement dates (`expiry_dates`), the symbols of the contracts and the number of days to roll the contract over (defaulting to five). The comments below explain the code:

```python
def futures_rollover_weights(start_date, expiry_dates,
    contracts, rollover_days=5):
    """This constructs a pandas DataFrame that contains weights
    (between 0.0 and 1.0) of contract positions to hold in order to
    carry out a rollover of rollover_days prior to the expiration of
    the earliest contract. The matrix can then be 'multiplied' with
    another DataFrame containing the settle prices of each
    contract in order to produce a continuous time series
    futures contract."""

    # Construct a sequence of dates beginning
    # from the earliest contract start date to the end
    # date of the final contract
    dates = pd.date_range(start_date, expiry_dates[-1], freq='B')

    # Create the 'roll weights' DataFrame that will store the multipliers for
    # each contract (between 0.0 and 1.0)
    roll_weights = pd.DataFrame(np.zeros((len(dates), len(contracts))),
                                index=dates, columns=contracts)
    prev_date = roll_weights.index[0]

    # Loop through each contract and create the specific weightings for
    # each contract depending upon the settlement date and rollover_days
    for i, (item, ex_date) in enumerate(expiry_dates.iteritems()):
        if i < len(expiry_dates) - 1:
            roll_weights.ix[prev_date:ex_date - pd.offsets.BDay(), item] = 1
            roll_rng = pd.date_range(end=ex_date - pd.offsets.BDay(),
                                     periods=rollover_days + 1, freq='B')

            # Create a sequence of roll weights (i.e. [0.0,0.2,...,0.8,1.0]
            # and use these to adjust the weightings of each future
            decay_weights = np.linspace(0, 1, rollover_days + 1)
            roll_weights.ix[roll_rng, item] = 1 - decay_weights
            roll_weights.ix[roll_rng,
                expiry_dates.index[i+1]] = decay_weights
        else:
            roll_weights.ix[prev_date:, item] = 1
        prev_date = ex_date
    return roll_weights
```

Now that the weighting matrix has been produced, it is possible to apply this to the individual time series. The main function downloads the near and far contracts, creates a single DataFrame for both, constructs the rollover weighting matrix and then finally produces a continuous series of both prices, appropriately weighted:

```
if __name__ == "__main__":
    # Download the current Front and Back (near and far) futures contracts
    # for WTI Crude, traded on NYMEX, from Quandl.com. You will need to
    # adjust the contracts to reflect your current near/far contracts
    # depending upon the point at which you read this!
    wti_near = Quandl.get("OFDP/FUTURE_CLF2014")
    wti_far = Quandl.get("OFDP/FUTURE_CLG2014")
    wti = pd.DataFrame({'CLF2014': wti_near['Settle'],
                        'CLG2014': wti_far['Settle']}, index=wti_far.index)

    # Create the dictionary of expiry dates for each contract
    expiry_dates = pd.Series(
        {'CLF2014': datetime.datetime(2013, 12, 19),
         'CLG2014': datetime.datetime(2014, 2, 21)}).order()

    # Obtain the rollover weighting matrix/DataFrame
    weights = futures_rollover_weights(wti_near.index[0],
                                       expiry_dates, wti.columns)

    # Construct the continuous future of the WTI CL contracts
    wti_cts = (wti * weights).sum(1).dropna()

    # Output the merged series of contract settle prices
    print(wti_cts.tail(60))
```

The output is as follows:

```
2013-10-14    102.230
2013-10-15    101.240
2013-10-16    102.330
2013-10-17    100.620
2013-10-18    100.990
2013-10-21     99.760
2013-10-22     98.470
2013-10-23     97.000
2013-10-24     97.240
2013-10-25     97.950
..
```

```
..
2013-12-24     99.220
2013-12-26     99.550
2013-12-27    100.320
2013-12-30     99.290
2013-12-31     98.420
2014-01-02     95.440
2014-01-03     93.960
2014-01-06     93.430
2014-01-07     93.670
2014-01-08     92.330
Length: 60, dtype: float64
```

It can be seen that the series is now continuous across the two contracts. This can be extended to handle multiple deliveries across a variety of years, depending upon your backtesting needs.

Part IV

Modelling

Chapter 9

Statistical Learning

The goal of the Modelling section within this book is to provide a robust quantitative framework for identifying relationships in financial market data that can be exploited to generate profitable trading strategies. The approach that will be utilised is that of *Statistical Learning*. This chapter describes the philosophy of statistical learning and associated techniques that can be used to create quantitative models for financial trading.

9.1 What is Statistical Learning?

Before discussing the theoretical aspects of statistical learning it is appropriate to consider an example of a situation from quantitative finance where such techniques are applicable. Consider a quantitative fund that wishes to make long term predictions of the S&P500 stock market index. The fund has managed to collect a substantial amount of *fundamental data* associated with the companies that constitute the index. Fundamental data includes *price-earnings ratio* or *book value*, for instance. How should the fund go about using this data to make predictions of the index in order to create a trading tool? Statistical learning provides one such approach to this problem.

In a more quantitative sense we are attempting to model the behaviour of an *outcome* or *response* based on a set of *predictors* or *features* assuming a relationship between the two. In the above example the stock market index value is the response and the fundamental data associated with the constituent firms are the predictors.

This can be formalised by considering a response Y with p different features $x_1, x_2, ..., x_p$. If we utilise *vector notation* then we can define $X = (x_1, x_2, ..., x_p)$, which is a vector of length p. Then the model of our relationship is given by:

$$Y = f(X) + \epsilon \tag{9.1}$$

Where f is an unknown function of the predictors and ϵ represents an *error* or *noise term*. Importantly, ϵ is not dependent on the predictors and has a mean of zero. This term is included to represent information that is not considered within f. Thus we can return to the stock market index example to say that Y represents the value of the S&P500 whereas the x_i components represent the values of individual fundamental factors.

The goal of statistical learning is to *estimate* the form of f based on the observed data and to evaluate how accurate those estimates are.

9.1.1 Prediction and Inference

There are two general tasks that are of interest in statistical learning - *prediction* and *inference*.

Prediction is concerned with predicting a response Y based on a *newly observed* predictor, X. If the model relationship has been determined then it is simple to predict the response using an estimate for f to produce an estimate for the response:

$$\hat{Y} = \hat{f}(X) \tag{9.2}$$

The functional form of f is often unimportant in a prediction scenario assuming that the estimated responses are close to the true responses and is thus accurate in its predictions. Different estimates of f will produce various accuracies of the estimates of Y. The error associated with having a poor estimate $\hat{f}$ of f is called the *reducible error*. Note that there is always a degree of *irreducible error* because our original specification of the problem included the ϵ error term. This error term encapsulates the unmeasured factors that may affect the response Y. The approach taken is to try and minimise the reducible error with the understanding that there will always be an upper limit of accuracy based on the irreducible error.

Inference is concerned with the situation where there is a need to understand the relationship between X and Y and hence its exact form must be determined. One may wish to identify important predictors or determine the relationship between individual predictors and the response. One could also ask if the relationship is *linear* or *non-linear*. The former means the model is likely to be more interpretable but at the expense of potentially worse predictability. The latter provides models which are generally more predictive but are sometimes less interpretable. Hence a trade-off between *predictability* and *interpretability* often exists.

In this book we are less concerned with inference models since the actual form of f is not as important as its ability to make accurate predictions. Hence a large component of the Modelling section within the book will be based on predictive modelling. The next section deals with how we go about constructing an estimate $\hat{f}$ for f.

9.1.2 Parametric and Non-Parametric Models

In a statistical learning situation it is often possible to construct a set of tuples of predictors and responses of the form $\{(X_1, Y_1), (X_2, Y_2), ..., (X_N, Y_N)\}$, where X_j refers to the jth predictor vector and not the jth component of a particular predictor vector (that is denoted by x_j). A data set of this form is known as *training data* since it will be used to *train* a particular statistical learning method on how to generate $\hat{f}$. In order to actually estimate f we need to find a $\hat{f}$ that provides a reasonable approximation to a particular Y under a particular predictor X. There are two broad categories of statistical models that allow us to achieve this. They are known as *parametric* and *non-parametric* models.

Parametric Models

The defining feature of parametric methods is that they require the *specification* or *assumption* of the form of f. This is a modelling decision. The first choice is whether to consider a linear or non-linear model. Let's consider the simpler case of a linear model. Such a model reduces the problem from estimation of some unknown function of dimension p to that of estimating a coefficient vector $\beta = (\beta_0, \beta_1, ..., \beta_p)$ of length $p + 1$.

Why $p + 1$ and not p? Since linear models can be *affine*, that is they may not pass through the origin when creating a "line of best fit", a coefficient is required to specify the "intercept". In a one-dimensional linear model (regression) setting this is often represented as α. For our multi-dimensional linear model, where there are p predictors, we need an additional value β_0 to represent our intercept and hence there are $p + 1$ components in our $\hat{\beta}$ estimate of β.

Now that we have specified a (linear) functional form of f we need to *train* it. "Training" in this instance means finding an estimate for β such that:

$$Y \approx \hat{\beta}^T X = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p \tag{9.3}$$

In the linear setting we can use an algorithm such as *ordinary least squares* (OLS) but other methods are available as well. It is far simpler to estimate β than fit a (potentially non-linear) f. However, by choosing a parametric linear approach our estimate $\hat{f}$ is unlikely to be replicating the true form of f. This can lead to poor estimates because the model is not *flexible* enough.

A potential remedy is to consider adding more parameters, by choosing alternate forms for $\hat{f}$. Unfortunately if the model becomes too flexible it can lead to a very dangerous situation known as *overfitting*, which we will discuss at length in subsequent chapters. In essence, the model follows the noise too closely and not the signal.

Non-Parametric Models

The alternative approach is to consider a non-parametric form of $\hat{f}$. The benefit is that it can potentially fit a wider range of possible forms for f and is thus more flexible. Unfortunately non-parametric models suffer from the need to have an extensive amount of observational data points, often far more than in a parametric settings. In addition non-parametric methods are also prone to overfitting if not treated carefully, as described above.

Non-parametric models may seem like a natural choice for quantitative trading models as there is seemingly an abundance of (historical) data on which to apply the models. However, the methods are not always optimal. While the increased flexibility is attractive for modelling the non-linearities in stock market data it is very easy to overfit the data due to the poor signal/noise ratio found in financial time series.

Thus a "middle-ground" of considering models with some degree of flexibility is preferred. We will discuss such problems in the chapter on Optimisation later in the book.

9.1.3 Supervised and Unsupervised Learning

A distinction is often made in statistical machine learning between *supervised* and *unsupervised* methods. In this book we will almost exclusively be interested in supervised techniques, but unsupervised techniques are certainly applicable to financial markets.

A supervised model requires that for each predictor vector X_j there is an associated response Y_j. The "supervision" of the procedure occurs when the model for f is *trained* or *fit* to this particular data. For example, when fitting a linear regression model, the OLS algorithm is used to train it, ultimately producing an estimate $\hat{\beta}$ to the vector of regression coefficients, β.

In an unsupervised model there is no corresponding response Y_j for any particular predictor X_j. Hence there is nothing to "supervise" the training of the model. This is clearly a much more challenging environment for an algorithm to produce results as there is no form of "fitness function" with which to assess accuracy. Despite this setback, unsupervised techniques are extremely powerful. They are particularly useful in the realm of *clustering*.

A parametrised clustering model, when provided with a parameter specifying the number of clusters to identify, can often discern unanticipated relationships within data that might not otherwise have been easily determined. Such models are generally fall within the domain of *business analytics* and *consumer marketing optimisation* but they do have uses within finance, particularly in regards to assessing clustering within volatility, for instance.

This book will predominantly concentrate on supervised learning methods since there is a vast amount of historical data on which to train such models.

9.2 Techniques

Statistical machine learning is a vast interdisciplinary field, with many disparate research areas. The remainder of this chapter will consider the techniques most relevant to quantitative finance and algorithmic trading in particular.

9.2.1 Regression

Regression refers to a broad group of supervised machine learning techniques that provide both predictive and inferential capabilities. A significant portion of quantitative finance makes use of regression techniques and thus it is essential to be familiar with the process. Regression tries to model the relationship between a dependent variable (response) and a set of independent variables (predictors). In particular, the goal of regression is to ascertain the change in a response, when one of the independent variables changes, under the assumption that the remaining independent variables are kept fixed.

The most widely known regression technique is *Linear Regression*, which assumes a linear relationship between the predictors and the response. Such a model leads to parameter estimates (usually denoted by the vector $\hat{\beta}$) for the linear response to each predictor. These parameters are estimated via a procedure known as *ordinary least squares* (OLS). Linear regression can be used both for prediction and inference.

In the former case a new value of the predictor can be added (without a corresponding response) in order to *predict* a new response value. For instance, consider a linear regression model used to predict the value of the S&P500 in the following day, from price data over the last five days. The model can be fitted using OLS across historical data. Then, when new market data arrive for the S&P500 it can be input into the model (as a predictor) to generate a predicted response for tomorrow's daily price. This can form the basis of a simplistic trading strategy.

In the latter case (inference) the strength of the relationship between the response and each predictor can be assessed in order to determine the subset of predictors that have an effect on the response. This is more useful when the goal is to understand *why* the response varies, such as in a marketing study or clinical trial. Inference is often less useful to those carrying out algorithmic trading, as the quality of the prediction is fundamentally more important than the underlying relationship. That being said, one should not solely rely on the "black box" approach due to the prevalence of over-fitting to noise in the data.

Other techniques include *Logistic Regression*, which is designed to predict a *categorical* response (such as "UP", "DOWN", "FLAT") as opposed to a *continuous* response (such as a stock market price). This technically makes it a *classification tool* (see below), but it is usually grouped under the banner of regression. A general statistical procedure known as *Maximum Likelihood Estimation* (MLE) is used to estimate the parameter values of a logistic regression.

9.2.2 Classification

Classification encompasses supervised machine learning techniques that aim to *classify* an *observation* (similar to a predictor) into a set of pre-defined categories, based on features associated with the observation. These categories can be un-ordered, e.g. "red", "yellow", "blue" or ordered, e.g. "low", "medium", "high". In the latter case such categorical groups are known as *ordinals*. Classification algorithms - *classifiers* - are widely used in quantitative finance, especially in the realm of market direction prediction. In this book we will be studying classifiers extensively.

Classifiers can be utilised in algorithmic trading to predict whether a particular time series will have positive or negative returns in subsequent (unknown) time periods. This is similar to a regression setting except that the actual value of the time series is not being predicted, rather its direction. Once again we are able to use continuous predictors, such as prior market prices as observations. We will consider both linear and non-linear classifiers, including Logistic Regression, Linear/Quadratic Discriminant Analysis, Support Vector Machines (SVM) and Artificial Neural Networks (ANN). *Note that some of the previous methods can actually be used in a regression setting also.*

9.2.3 Time Series Models

A key component in algorithmic trading is the treatment and prediction of *financial time series*. Our goal is generally to predict future values of time series based on prior values or external factors. Thus time series modelling can be seen as a mixed-subset of regression and classification. Time series models differ from non-temporal models because the models make deliberate use of the *temporal ordering* of the series. Thus the predictors are often based on past or current values, while the responses are often future values to be predicted.

There is a large literature on differing time series models. There are two broad families of time series models that interest us in algorithmic trading. The first set are the linear *autoregressive integrated moving average* (ARIMA) family of models, which are used to model the variations in the absolute value of a time series. The other family of time series are the *autoregressive* *conditional heteroskedasticity* (ARCH) models, which are used to model the variance (i.e. the volatility) of time series over time. ARCH models use previous values (volatilities) of the time series to predict future values (volatilities). This is in contrast to *stochastic volatility* models, which utilise more than one stochastic time series (i.e. multiple stochastic differential equations) to model volatility.

All of the raw historical price time series are *discrete* in that they contain finite values. In the field of quantitative finance it is common to study *continuous* time series models. In particular, the famous *Geometric Brownian Motion*, the *Heston Stochastic Volatility* model and the *Ornstein-Uhlenbeck* model all represent continuous time series with differing forms of stochastic behaviour. We will utilise these time series models in subsequent chapters to attempt to characterise the behaviour of financial time series in order to exploit their properties to create viable trading strategies.

Chapter 10

Time Series Analysis

In this chapter we are going to consider statistical tests that will help us identify price series that possess trending or mean-reverting behaviour. If we can identify such series statistically then we can capitalise on this behaviour by forming momentum or mean-reverting trading strategies.

In later chapters we will use these statistical tests to help us identify candidate time series and then create algorithmic strategies around them.

10.1 Testing for Mean Reversion

One of the key quantitative trading concepts is **mean reversion**. This process refers to a time series that displays a tendency to revert to a historical mean value. Such a time series can be exploited to generate trading strategies as we enter the market when a price series is far from the mean under the expectation that the series will return to a mean value, whereby we exit the market for a profit. Mean-reverting strategies form a large component of the *statistical arbitrage* quant hedge funds. In later chapters we will create both intraday and interday strategies that exploit mean-reverting behaviour.

The basic idea when trying to ascertain if a time series is mean-reverting is to use a statistical test to see if it differs from the behaviour of a *random walk*. A random walk is a time series where the next directional movement is completely independent of any past movements - in essence the time series has no "memory" of where it has been. A mean-reverting time series, however, is different. The change in the value of the time series in the next time period is proportional to the current value. Specifically, it is proportional to the difference between the mean historical price and the current price.

Mathematically, such a (continuous) time series is referred to as an **Ornstein-Uhlenbeck** process. If we can show, statistically, that a price series behaves like an Ornstein-Uhlenbeck series then we can begin the process of forming a trading strategy around it. Thus the goal of this chapter is to outline the statistical tests necessary to identify mean reversion and then use Python libraries (in particular *statsmodels*) in order to implement these tests. In particular, we will study the concept of **stationarity** and how to test for it.

As stated above, a *continuous* mean-reverting time series can be represented by an Ornstein-Uhlenbeck stochastic differential equation:

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t \tag{10.1}$$

Where θ is the rate of reversion to the mean, μ is the mean value of the process, σ is the variance of the process and W_t is a Wiener Process or Brownian Motion.

This equation essentially states that the change of the price series in the next continuous time period is proportional to the difference between the mean price and the current price, with the addition of Gaussian noise.

We can use this equation to motivate the definition of the Augmented Dickey-Fuller Test, which we will now describe.

10.1.1 Augmented Dickey-Fuller (ADF) Test

The ADF test makes use of the fact that if a price series possesses mean reversion, then the next price level will be proportional to the current price level. Mathematically, the ADF is based on the idea of testing for the presence of a **unit root** in an **autoregressive** time series sample.

We can consider a model for a time series, known as a *linear lag model of order p*. This model says that the change in the value of the time series is proportional to a constant, the time itself and the previous p values of the time series, along with an error term:

$$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \delta_1 \Delta y_{t-1} + \cdots + \delta_{p-1} \Delta y_{t-p+1} + \epsilon_t \tag{10.2}$$

Where α is a constant, β represents the coefficient of a temporal trend and $\Delta y_t = y(t) - y(t-1)$. The role of the ADF hypothesis test is to ascertain, statistically, whether $\gamma = 0$, which would indicate (with $\alpha = \beta = 0$) that the process is a random walk and thus non mean reverting. Hence we are testing for the *null hypothesis* that $\gamma = 0$.

If the hypothesis that $\gamma = 0$ can be rejected then the following movement of the price series is proportional to the current price and thus it is unlikely to be a random walk. This is what we mean by a "statistical test".

So how is the ADF test carried out?

- Calculate the *test statistic*, DF_τ, which is used in the decision to reject the null hypothesis
- Use the *distribution* of the test statistic (calculated by Dickey and Fuller), along with the critical values, in order to decide whether to reject the null hypothesis

Let's begin by calculating the test statistic (DF_τ). This is given by the *sample* proportionality constant $\hat{\gamma}$ divided by the *standard error* of the sample proportionality constant:

$$DF_\tau = \frac{\hat{\gamma}}{SE(\hat{\gamma})} \tag{10.3}$$

Now that we have the test statistic, we can use the distribution of the test statistic calculated by Dickey and Fuller to determine the rejection of the null hypothesis for any chosen percentage critical value. The test statistic is a negative number and thus in order to be significant beyond the critical values, the number must be smaller (i.e. more negative) than these values.

A key practical issue for traders is that any constant long-term drift in a price is of a much smaller magnitude than any short-term fluctuations and so the drift is often assumed to be zero ($\beta = 0$) for the linear lag model described above.

Since we are considering a lag model of order p, we need to actually set p to a particular value. It is usually sufficient, for trading research, to set $p = 1$ to allow us to reject the null hypothesis. However, note that this technically introduces a parameter into a trading model based on the ADF.

To calculate the Augmented Dickey-Fuller test we can make use of the pandas and statsmodels libraries. The former provides us with a straightforward method of obtaining Open-High-Low-Close-Volume (OHLCV) data from Yahoo Finance, while the latter wraps the ADF test in a easy to call function. This prevents us from having to calculate the test statistic manually, which saves us time.

We will carry out the ADF test on a sample price series of Amazon stock, from 1st January 2000 to 1st January 2015.

Here is the Python code to carry out the test:

```python
from __future__ import print_function

# Import the Time Series library
import statsmodels.tsa.stattools as ts

# Import Datetime and the Pandas DataReader
```

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

# Download the Amazon OHLCV data from 1/1/2000 to 1/1/2015
amzn = web.DataReader("AMZN", "yahoo", datetime(2000,1,1), datetime(2015,1,1))

# Output the results of the Augmented Dickey-Fuller test for Amazon
# with a lag order value of 1
ts.adfuller(amzn['Adj Close'], 1)
```

Here is the output of the Augmented Dickey-Fuller test for Amazon over the period. The first value is the calculated test-statistic, while the second value is the *p-value*. The fourth is the number of data points in the sample. The fifth value, the dictionary, contains the critical values of the test-statistic at the 1, 5 and 10 percent values respectively.

```
(0.049177575166452235,
 0.96241494632563063,
 1,
 3771,
 {'1%': -3.4320852842548395,
  '10%': -2.5671781529820348,
  '5%': -2.8623067530084247},
 19576.116041473877)
```

Since the calculated value of the test statistic is larger than any of the critical values at the 1, 5 or 10 percent levels, we cannot reject the null hypothesis of $\gamma = 0$ and thus we are unlikely to have found a mean reverting time series. This is in line with our tuition as most equities behave akin to Geometric Brownian Motion (GBM), i.e. a random walk.

This concludes how we utilise the ADF test. However, there are alternative methods for detecting mean-reversion, particularly via the concept of **stationarity**, which we will now discuss.

10.2 Testing for Stationarity

A time series (or stochastic process) is defined to be **strongly stationary** if its *joint probability distribution* is invariant under translations in time or space. In particular, and of key importance for traders, the mean and variance of the process do not change over time or space and they each do not follow a trend.

A critical feature of stationary price series is that the prices within the series diffuse from their initial value at a rate slower than that of a GBM. By measuring the rate of this diffusive behaviour we can identify the nature of the time series and thus detect whether it is mean-reverting.

We will now outline a calculation, namely the Hurst Exponent, which helps us to characterise the stationarity of a time series.

10.2.1 Hurst Exponent

The goal of the Hurst Exponent is to provide us with a scalar value that will help us to identify (within the limits of statistical estimation) whether a series is mean reverting, random walking or trending.

The idea behind the Hurst Exponent calculation is that we can use the variance of a log price series to assess the rate of diffusive behaviour. For an arbitrary time lag τ, the variance of τ is given by:

$$\mathrm{Var}(\tau) = \langle |\log(t+\tau) - \log(t)|^2 \rangle \tag{10.4}$$

Where the brackets $\langle$ and $\rangle$ refer to the average over all values of t.

The idea is to compare the rate of diffusion to that of a GBM. In the case of a GBM, at large times (i.e. when τ is large) the variance of τ is proportional to τ:

$$\langle |\log(t+\tau) - \log(t)|^2 \rangle \sim \tau \tag{10.5}$$

If we find behaviour that differs from this relation, then we have identified either a trending or a mean-reverting series. The key insight is that if any sequential price movements possess non-zero correlation (known as autocorrelation) then the above relationship is not valid. Instead it can be modified to include an exponent value "$2H$", which gives us the Hurst Exponent value H:

$$\langle |\log(t+\tau) - \log(t)|^2 \rangle \sim \tau^{2H} \tag{10.6}$$

Thus it can be seen that if $H = 0.5$ we have a GBM, since it simply becomes the previous relation. However if $H \neq 0.5$ then we have trending or mean-reverting behaviour. In particular:

- $H < 0.5$ - The time series is mean reverting
- $H = 0.5$ - The time series is a Geometric Brownian Motion
- $H > 0.5$ - The time series is trending

In addition to characterisation of the time series the Hurst Exponent also describes the extent to which a series behaves in the manner categorised. For instance, a value of H near 0 is a highly mean reverting series, while for H near 1 the series is strongly trending.

To calculate the Hurst Exponent for the Amazon price series, as utilised above in the explanation of the ADF, we can use the following Python code:

```python
from __future__ import print_function

from numpy import cumsum, log, polyfit, sqrt, std, subtract
from numpy.random import randn

def hurst(ts):
    """Returns the Hurst Exponent of the time series vector ts"""
    # Create the range of lag values
    lags = range(2, 100)

    # Calculate the array of the variances of the lagged differences
    tau = [sqrt(std(subtract(ts[lag:], ts[:-lag]))) for lag in lags]

    # Use a linear fit to estimate the Hurst Exponent
    poly = polyfit(log(lags), log(tau), 1)

    # Return the Hurst exponent from the polyfit output
    return poly[0]*2.0

# Create a Gometric Brownian Motion, Mean-Reverting and Trending Series
gbm = log(cumsum(randn(100000))+1000)
mr = log(randn(100000)+1000)
tr = log(cumsum(randn(100000)+1)+1000)

# Output the Hurst Exponent for each of the above series
# and the price of Amazon (the Adjusted Close price) for
# the ADF test given above in the article
print("Hurst(GBM):   %s" % hurst(gbm))
print("Hurst(MR):    %s" % hurst(mr))
print("Hurst(TR):    %s" % hurst(tr))
```

```
# Assuming you have run the above code to obtain 'amzn'!
print("Hurst(AMZN):  %s" % hurst(amzn['Adj Close']))
```

The output from the Hurst Exponent Python code is given below:

```
Hurst(GBM):   0.502051910931
Hurst(MR):    0.000166110248967
Hurst(TR):    0.957701001252
Hurst(AMZN):  0.454337476553
```

From this output we can see that the GBM possesses a Hurst Exponent, H, that is almost exactly 0.5. The mean reverting series has H almost equal to zero, while the trending series has H close to 1.

Interestingly, Amazon has H also close to 0.5 indicating that it is similar to a GBM, at least for the sample period we're making use of!

10.3 Cointegration

It is actually very difficult to find a tradable asset that possesses mean-reverting behaviour. Equities broadly behave like GBMs and hence render the mean-reverting trade strategies relatively useless. However, there is nothing stopping us from creating a *portfolio* of price series that is stationary. Hence we can apply mean-reverting trading strategies to the portfolio.

The simplest form of mean-reverting trade strategies is the classic "pairs trade", which usually involves a dollar-neutral long-short pair of equities. The theory goes that two companies in the same sector are likely to be exposed to similar market factors, which affect their businesses. Occasionally their relative stock prices will diverge due to certain events, but will revert to the long-running mean.

Let's consider two energy sector equities Approach Resources Inc given by the ticker AREX and Whiting Petroleum Corp given by the ticker WLL. Both are exposed to similar market conditions and thus will likely have a stationary pairs relationship. We are now going to create some plots, using pandas and the Matplotlib libraries to demonstrate the cointegrating nature of AREX and WLL. The first plot (Figure 10.1) displays their respective price histories for the period Jan 1st 2012 to Jan 1st 2013.

If we create a scatter plot of their prices, we see that the relationship is broadly linear (see Figure 10.2) for this period.

The pairs trade essentially works by using a linear model for a relationship between the two stock prices:

$$y(t) = \beta x(t) + \epsilon(t) \qquad (10.7)$$

Where $y(t)$ is the price of AREX stock and $x(t)$ is the price of WLL stock, both on day t.

If we plot the residuals $\epsilon(t) = y(t) - \beta x(t)$ (for a particular value of β that we will determine below) we create a new time series that, at first glance, looks relatively stationary. This is given in Figure 10.3.

We will describe the code for each of these plots below.

10.3.1 Cointegrated Augmented Dickey-Fuller Test

In order to statistically confirm whether this series is mean-reverting we could use one of the tests we described above, namely the Augmented Dickey-Fuller Test or the Hurst Exponent. However, neither of these tests will actually help us determine β, the hedging ratio needed to form the linear combination, they will only tell us whether, for a particular β, the linear combination is stationary.

This is where the Cointegrated Augmented Dickey-Fuller (CADF) test comes in. It determines the optimal hedge ratio by performing a linear regression against the two time series and then tests for stationarity under the linear combination.