Escuela Técnica Superior de Ingeniería de Sistemas Informáticos

Departamento de Matemática Aplicada a las
Tecnologías de la Información y las Telecomunicaciones

Universidad Politécnica de Madrid

Ingeniería del Software

Bachelor's Degree Project

Design and Development of a Functional High-Frequency Trading Algorithm

Authors: Luis Adolfo González Corujo **y** Diego González Franco

Supervisor: Francisco Gómez Martín

July 2019

Contents

Abstract

Throughout this project, a complete functional system capable of automating trading actions in the Forex market such as price analysis and order management has been studied, developed and implemented.

This core of the system is able to receive a large amount of market prices, analyze them and determine before each of these new prices the action that has to be carried out: buy an order or sell an order previously bought. The algorithm can be used either to perform tests using a database of historical prices, or to trade live connecting to a broker platform. In much of the development, we have relied on research carried out over several decades of study, which culminated in a proof-of-concept algorithm called "The Alpha Engine".

From the beginning, our main objective has been to develop a similar system to "The Alpha Engine" that could be used in the real world by any user, and to introduce elements that this model lacks in order to improve its efficiency and operation.

Throughout the chapters of this report, we explain the developments carried out in the logical module of the algorithm: from a new method to model the Forex market price series to a system to control the risk that is assumed in each of the open actions, explaining also the trading strategy and the computation of a statistical indicator of the market liquidity that manages the aggressiveness with which the algorithm invests, based on the knowledge of the recent behaviour of the market. In addition, we explain how the architecture of the system has been designed and defined for the system to be used by any user who has no more than an Internet connection.

Resumen

A lo largo de este proyecto, ha sido estudiado, desarrollado e implementado un sistema completamente funcional capaz de automatizar tareas de 'Trading' en el mercado de divisas Forex como son analizar los precios y gestionar las acciones abiertas.

El núcleo del sistema es capaz de recibir una gran cantidad precios de mercado, analizarlos y determinar ante cada uno de estos precios la acción que debe ser llevada a cabo: comprar una acción o vender una previamente abierta. El algoritmo puede ser utilizado tanto para realizar pruebas por medio de una base de datos de precios histórica, como para comerciar en el mercado real conectándolo a una plataforma 'broker'. En mayor parte del desarrollo, nos hemos guiado por investigaciones llevadas a cabo durante varias décadas de estudio, las cuales culminaron en un algoritmo prueba de concepto llamado"The Alpha Engine".

Desde el principio, nuestro objetivo principal ha sido desarrollar un sistema similar a "The Alpha Engine" que pueda ser usado en el mundo real por cualquier usuario que así lo desee, así como introducir nuevos elementos de los que este modelo carece para así mejorar su eficiencia y funcionamiento.

Durante los capítulos de esta memoria, se explican los desarrollos de la parte lógica del algoritmo: desde un nuevo método para modelar la línea de tiempo del mercado Forex hasta un sistema para controlar el riesgo que se asume en cada una de las acciones abiertas, pasando por la estrategia de 'trading' y por el cálculo de indicador estadístico de la liquidez del mercado que controla la agresividad con la que el algoritmo invierte en función del conocimiento que se tiene del comportamiento reciente del mercado. Además, se explica cómo ha sido diseada y definida una arquitectura con la que este algoritmo desarrollado podrá ser utilizado por cualquier usuario que únicamente disponga de una conexión a internet.

Chapter 1

Introduction to Algorithmic Trading

1.1 Introduction

Could you imagine a universe in which all tasks and jobs, even the smallest ones, were performed by robots? Let's imagine that your smartphone wakes you up in the morning, you step on the floor that your heating system has warmed up for you a few minutes before and you go to your smart bathroom that already has prepared you a hot shower (or a fresh one if it is summer). Meanwhile, your smart kitchen is preparing you your breakfast. Exactly that breakfast that you like so much. Imagine that your smart car takes you to your job while you are just paying attention to the news or that selection of music that some device from the non-so-far future has selected specifically for you. Imagine that all your problems were solved by a computer or a tablet. Imagine that you could buy anything, even the most absurd things, through more and more intelligent programs that can suggest you exactly what you want the most. Even if you didn't know what that was. Imagine that you had a device that could get ahead and solve all your needs and problems, even before they appear. It doesn't sound bad, does it?

Algorithmic Trading is a method of operating in financial markets that use computer programs. These programs follow certain rules defined with the objective of executing operations of purchase or sale of financial instruments. The main benefit of this trading method is that, theoretically, it can generate profit with such precision and speed impossible to reach for a human.

Algorithmic trading dates back to the beginning of the 1970s, when it began to computerize the flow of orders in the financial markets. This computerization of market orders allowed the development of systems such as DOT (also known as SuperDOT), which was used in the New York Stock Exchange (NYSE) until 2009. DOT was an order routing system in which orders were sent directly to the screens of specialists, thus avoiding the intervention of brokers [1]. The specialists were helped by a computerized system called OARS (Opening Automated Reporting Service) with which they could detect imbalances in the orders before the market opened, helping them determine the opening price [2]. These two combined systems allowed to gain in speed and precision, so managing to reduce the number of errors and thus eliminating human intervention in the order handling process.

In the 1980s, computational advances caused by algorithmic trading began to be widely used in two of the most important markets in the United States of America: The Standard & Poor's 500 equity and The Standard & Poor's 500 futures markets.

In these markets, traders used to buy or sell shares to futures and the trading program automatically entered them into the NYSE's electronic order routing system until the time between the entry price and the current price of the shares were big enough to obtain benefits. This and other strategies emerged in that decade were grouped and started to be known as "program trading" [3].

Many people pointed to program trading as the culpable of aggravating or even starting 1987's stock market crisis, as we can see in the following citation from the Report of the Presidential Task Force on Market Mechanisms, submitted to the President of the United States, The Secretary of the Treasury and The Chairman of the Federal Reserve Board (commonly known as the Brady Report) published in January 1988 [4]:

> *Among technical factors, portfolio insurance, stock index arbitrage, program trading (obviously not mutually exclusive responses), specialist system mechanics, and poor capitalization of specialists were the five most frequently cited reasons for the market's decline on October 19.*

According to this report [4], at that time, several specialists came to suggest limiting or banning program trading. Most of the support for this measure came from banks and securities firms, while pension funds were strongly opposed to this possible action. Finally, the measure was not carried out as they realized that it would be an inappropriate and unhelpful action.

To this day, the impact of algorithmic and computer-driven trading on market downturns and crises is not clear and is widely discussed in the academic community.

An important moment for the development of algorithmic trading comes at the end of the 80s and the beginning of the 90s when arising the fully electronic execution financial markets.

Another noteworthy fact is the change of decimalization in the United States, which caused the change in the minimum tick size from 1/16 dollar (US $ 0.0625) to US $ 0.01 per share in 2001 [5]. This may have encouraged algorithmic operations, since it changed the micro-structure of the market.

This greater liquidity of the market led institutional traders to divide orders according to computer algorithms so that they could execute orders at a better average price.

A key point in the history of algorithmic trading was the publication of the paper "Agent-Human Interactions in the Continuous Double Action" by a team of IBM researchers in 2001 at the International Joint Conference on Artificial Intelligence [6]. In that paper, they showed, by using an empirical method, how the use of two algorithmic strategies (IBM's MGD and Hewlett-Packard's ZIP) could surpass human traders in the financial markets:

Another notable difference is that the successful demonstration of machine superiority in the CDA and other common auctions could have a much more direct and powerful financial impact— one that might be measured in billions of dollars annually.

Due to the numerous advances in this field, trading algorithms have evolved and increasingly arise more and more in the recent years. Current algorithms are capable of analyzing several data sources. Among them, it is usual for the most advanced algorithms to analyze, in addition to the current price of the financial asset, values that provide information on the state or micro-structure of the market in question, such as opening or closing auctions, volatility auctions, suspension limits of species and even news published in real time in platforms such as Bloomberg. Some examples of the most used algorithms in recent times are Chameleon (developed by BNP Paribas), Stealth (developed by the Deutsche Bank), Sniper and Guerilla (developed by Credit Suisse), arbitrage, statistical arbitrage, trend following, and mean reversion [3].

In this context, and despite the multiple advances that have been made, a big disadvantage present in all these algorithms is the absence of a completely reliable and consistent framework that automates the entire trading process and manages to entirely put aside the human factor. This absence causes the so-called Trading Psychology to come into play in the current algorithmic trading [7].

Trading psychology refers to the emotions and mental state that can turn out to be a decisive factor to achieve success or failure in the negotiation of values. Trading psychology groups several aspects of the character and behavior of an individual that may influence their commercial actions. These behaviors can vary from greed, which often causes the investor to stay in a profitable trade for longer than is advisable in an effort to squeeze out the additional benefits; fear, which causes traders to close positions prematurely or refrain from taking risks due to concerns about large losses; or repentance, by which to trader may get into a trade after being missing out on it because of the stock moved too fast.

Trading psychology can be as important as other attributes such as knowledge, experience and the ability to determine the success of the trade. In algorithmic trading, trading psychology can influence, for example, the choice and configuration of the indicators, which is subject to the specific preference of the analyst or trader. In effect, investors mostly apply ad hoc rules that are not correctly analyzed or integrated into a broader context. Complex phenomena, such as changing liquidity levels as a function of time, are often neglected.

Fighting against this subjectivity when investing in the stock market has historically been one of the greatest challenges on the part of investors, as we can see from the following quote from Edgar Bonham-Carter, CEO of Jupiter Asset Management [8]:

Know yourself. Overcoming human instinct is the key to become an excellent investor.

Fortunately, there is an exception in the trading industry that fully relies on automatic trading algorithms and minimizes the human factor in the executions. This exception is

known as *high-frequency trading* [9]. The high-frequency trading, also known by its acronym HFT, is a type of trading that is done using powerful algorithms in an automated way to obtain market information and depending on this information, buy or sell financial securities. HFT is a technique that heavily relies on the bases of quantitative analysis. In other words, HFT uses financial mathematics to carry out market analysis. Generally speaking, investment positions open by HFT are held for very short periods of time, resulting in quick purchases or sales of assets. This means that throughout the day thousands, or even tens of thousands of operations, can be made.

By using mathematical models and algorithms to make decisions, high-frequency trading has managed to eliminate the human decision factor, causing the replacement of numerous broker-dealers. For this reason, high-frequency trading is so controversial and has been subject to numerous criticisms since its inception.

Existing high-frequency trading algorithms make decisions in milliseconds, so a misconfiguration can trigger large market movements without apparent cause. An example of this is the so-called "Flash Crash" of May 6, 2010, when the Dow Jones Industrial Average (DJIA) suffered its largest intra-day point drop ever, declining 1,000 points and dropping 10% in just 20 minutes before rising again [10]. A government investigation blamed a massive order that triggered a sell-off for the crash. As a result, several European countries came to propose curtailing or banning high-frequency trading algorithms due to concerns about volatility.

Due to the accuracy and speed required to operate in HFT, algorithms must be extremely reliable in order to guarantee profit. The accuracy required and complexity of the calculations that must be done internally are so high that currently no high-frequency trading model can be considered as perfect and completely reliable.

In this work, we carried out the study, development, implementation and improvement of an algorithm based on high-frequency trading, which, by using strong mathematical tools, tries to remove the main disadvantages of the existing algorithms. Specifically, we will develop an algorithm that will be robust to the market changes and that will need the minimum possible human intervention. Afterwards, we will show evidence of the execution and functioning of the algorithm using historical market data.

Chapter 2

Financial Markets

2.1 Looking for the best market to develop a trading algorithm

The first step to carry out the development of a high-frequency trading algorithm is to find the right market for its operation. For this reason, it must be taken into account that the characteristics of the different markets can benefit or impair the effectiveness of algorithmic trading.

In the first place, the market in which high-frequency trading is carried out must be as volatile as possible. The search for a volatile market, which is often seen as a negative factor, is due to the fact that algorithms of this type base their functioning on the continuous execution of mathematical operations which eventually provoke a response to changes in the market price. The faster and more widely the market price varies, the higher the amount of operations the algorithm will execute, and in case it is designed correctly, the more profit will be reaped.

Second, and in relation to the previous paragraph, it is convenient that the market price is updated as many times as possible during a given period of time. When performing the mathematical operations with each new price that arrives at the algorithm, the higher the frequency this value is updated, the more operations will be executed.

Third, it is optimal that the market in which the algorithm works can be accessed at any time of the day. This is due to the fact that in high-frequency trading algorithms all negotiation decisions, whether buying, selling with profit or getting rid of an order that has reached the maximum admitted loss, are taken by the computer. This factor makes it possible to take advantage of the fact that the algorithm can be working autonomously 24 hours a day without the need for a person to supervise operations.

Fourth, and in order to achieve the maximum profit, the costs of operating in the market should be minimal, since these trading algorithms are characterized by performing a high number of operations daily and in short periods of time. The lower the management costs derived from opening or closing an order, the greater the benefits.

Finally, and in order for a high-frequency trading algorithm to work properly, it is necessary for any transaction to be carried out in the shortest possible time, without the market price changing again in that time.

After taking into account all the above factors and making an analysis of the characteristics of the main markets in which it is possible to operate using algorithms, we conclude that, due to its nature, the optimal market for the proper functioning of a high-frequency trading algorithm is the Foreign Exchange Market, also known as Currency Market, and commonly abbreviated as Forex or FX.

2.2 Forex Market

The currency market concept has existed for thousands of years, practically since the invention of the money, when the need to exchange the currency of one region for another arose.

Today, the Forex can be considered as a large and complex connection through a computer network of agents that interact with each other and among which are mainly the following[11]:

- Large central banks such as the Deutsche Bank. These central banks are responsible for providing foreign currency liquidity so that they can be exchanged by the rest of the agents.
- Other non-central banks such as Citibank or Bank of America. The Forex is the market where these banks, of all sizes, trade currency with each other.
- Hedge funds and Investment managers like JP Morgan or Morgan Stanley. Investment managers exchange currencies for large accounts, such as pension funds, foundations and endowments.
- Large and diverse international Corporations. Firms engaged in importing and exporting conduct forex transactions to pay for goods and services.
- Institutional and retail traders. Although the volume of operations of this group is low compared to financial and emotional institutions, this volume is growing rapidly in recent years. Retail investors base currency trades on a combination of fundamental analysis (i.e., interest rate parity, inflation rates, and monetary policy expectations) and technical factors (i.e., support, resistance, technical indicators, price patterns).

The main purpose of this market is to bring together all international capital transactions of any kind, whether they come from commercial transactions, financial operations, central banks or simply speculative movements. In a very simplified way, in the Forex a certain currency of a country is bought paying with the currency of another. For this reason, Forex operations always involve two currencies, so traders will always work with pairs of currencies. To give an example, the most common and used pairs are the euro-dollar pair, the pound-dollar pair and the dollar-yen pair.

The technical characteristics of the Foreign Exchange Market make it an ideal environment for high-frequency operations.

First of all, Forex is, by a wide margin, the largest market and with the highest volume of daily operations in the world, reaching a volume of approximately $ 5.09 billion per day in April 2016 [12], which is a larger volume than all the other stock markets of the planet combined.

Second, the market prices of the currency pairs change practically every second. This rate of change, together with the large daily volume described in the previous paragraph, make Forex a market with external liquidity.

Another factor that favors this great liquidity is the fact that of the total volume traded in Forex, approximately 90% is considered speculative, which means that only 10% of the volume comes from commercial transactions or investments [13]. Therefore, the liquidity and depth of market that the Forex offers us is incomparable to any other financial market of the last one.

Unlike other financial markets where the prices of the assets are quoted in reference to specific values, the Forex exchange rates are symmetrical: the quotes are currencies in reference to other currencies. The symmetry of one currency against another one neutralizes the effects of market trends. Due to this property of symmetry, the Foreign Exchange Market is considered notoriously difficult to predict and negotiate profitably.

Finally, Forex is a market that is not physically centralized anywhere in the world, as it is in other markets such as the stock exchange, but is part of a large network of negotiation through international banks with each other. This fact makes it possible to operate on it 24 hours a day, 5 days a week, while closing only on weekends.

For all these reasons, we will focus on the study and analysis of the behavior of the Foreign Exchange Market, since its extreme liquidity and its symmetry make it the ideal environment for research and development of fully automated and algorithmic trading strategies.

Chapter 3

Developing a High-Frequency Trading Algorithm

3.1 The Intrinsic Time

3.1.1 History of the Intrinsic Time

In the previous section, we described the anatomy of the Forex market and explained why it is the best market to focus on for our high-frequency trading algorithm. Among other things, we mentioned that, unlike the stock market, the Forex opens 24 hours a day five days of the week. The fact that the Forex has this schedule causes that the activity of the market may vary enormously throughout the day.

Some events external to the Forex, such as an unexpected action by a central bank or news such as the results of political elections, may provoke the arise of strong trends in the market trading activity as a response. A clear example of this, and how chaotic can be for the health of the market, occurred when in 2011, in the splendour of the Great Crisis [14], the Swiss National Bank faced the risk of a massive flight of capital invested in the Eurozone by setting the minimum Exchange rate at CHF / EUR currency pair at 1.20 [15]. This action provoked very aggressive reactions in the market and during the following year, considerable losses for the SNB itself [16].

Consequently, it is understood that the financial time series in the Forex Market are unevenly spaced, which makes the flow of physical time discontinuous. In order to model these financial time series and to understand the behaviour of the market in a scientific way, multiple studies have been carried out by different researchers from 1960s onwards.

Firstly, Mandelbrot and Taylor in 1967 [18] analysed how the price changes over a fixed number of transactions, establishing that it might follow Gaussian distribution, and also how the price changes over a fixed time period, establishing that it might follow a Paretian distribution. Additionally, they introduced a **transaction clock** that ticks at every worldwide transaction.

At the end of the 80s, Stock [19] showed that there is evidence that postwar U.S. real

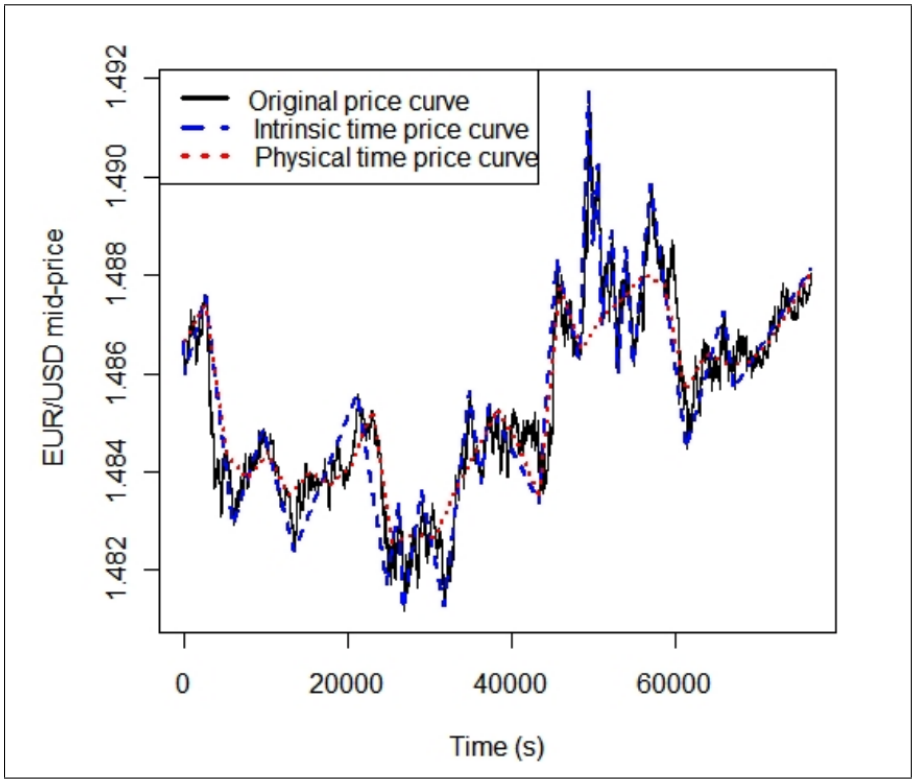


Figure 3.1: EUR/USD mid-prices time series defined by physical time and intrinsic time. The sampling period is over November 5th, 2009. For intrinsic time, the number of events is 30 determined by a threshold size of 0.1%. [17]

gross national product (GNP) evolves in a time scale other than calendar time. In particular, in their research, Stock introduced a new time scale defined by the speed of the aggregated market activity.

In 1995 a key point is reached, when Muller, Dacorogna, Davé, Pictet, Olsen and Ward in [20] propose the concept of **intrinsic time** as the cumulative sum of variable market activity and successfully apply it to the time scale of a Forex forecasting model.

Finally, in 1997, Guillaume, Olsen, Muller, Dacorogna, Pictet and Davé [21] introduce the **directional-change event** approach where they use intrinsic time in order to define time in price series, eliminating any irrelevant details of price evolution between intrinsic events.

All this research supposed an immense advance in the matter, since they allowed to define a new concept, the mentioned intrinsic time, that allow us to understand the time differently as opposed to the traditional concept of physical time. Additionally, in the context of the markets, this new concept of intrinsic time was used to model the financial price series, defining a new perspective, the directional-change approach, with which it is intended to understand the behavior of the market on a more successful way than using physical time.

Using as a basis all this research concerning the use of the concept of intrinsic time to

model the **price curve** of the financial markets, Aloud, Tsang, Olsen and Dupuis in [17] analyse and study the results of applying the directional-change approach to model the Forex market price series using real data. In detail, they use a data sample with high-frequency tick-by-tick price data of the currency pairs EUR / USD and EUR / CHF from January 2006 to December 2009. The results presented in this study indicate that the directional-change event approach may involve improving understanding of the behaviour of financial markets. Indeed, by correctly applying the concept of intrinsic time, we can model a price curve of the market in a better way than we would by using the traditional concept of physical time.

Due to the excellent results and, consequently, the large profit opportunity that these results suggest, we will use the intrinsic time approach as the main basis to model the price curve of the Forex market in our high-frequency trading algorithm.

In detail, for the algorithm to be developed to be able to perform an adequate analysis of the market price curve using the concept of intrinsic time, we will distinguish two main intrinsic events that will occur depending on how that price varies: directional change and overshoot. However, before explaining these events, we will have to describe several necessary previous concepts: market mode, short trading, long trading and threshold.

3.1.2 Previous Concepts

The first previous concept, **market mode**, is, broadly speaking, the trend that follows the market price since the last recorded event. In case that the last recorded event takes place at a higher price than the penultimate registered event, the market has a **mode up**. Otherwise, if the last recorded event takes place at a lower price than the penultimate event recorded, the market have a **mode down**; see Figure 3.2.

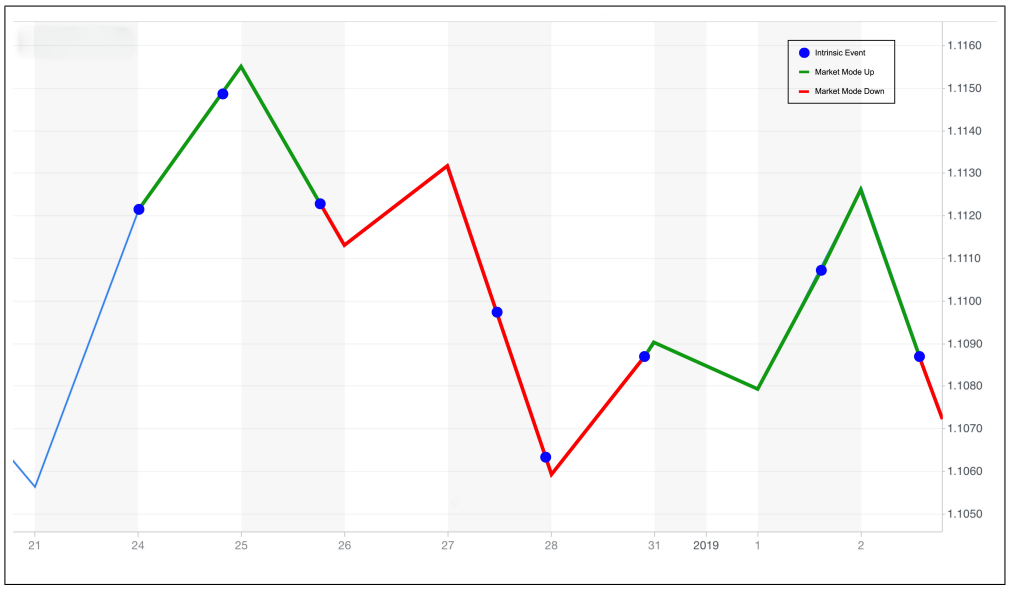


Figure 3.2: Market mode.

Furthermore, the terms long and short trading are concepts typical of the jargon of stock markets and trading. **Long trading** refers to investing in positions that are expected to go up in price. Long is the usual trading strategy that we understand by investing in the stock market, according to which, ideally, we will open positions at low values and close them when this value has risen to produce the expected benefits.

On the other hand, **short trading** refers to investing with the objective of making profits when the market falls. In short trading, we will open positions when we believe that the value of a currency has reached a maximum value from which we think it can only go down, and we will close it obtaining benefits once the value has dropped.

In order for the short trading to be possible, the investor asks to "borrow" an amount of a currency at a certain moment. When the investor wishes to close the position, he must return the amount of the currency he originally borrowed, making a profit in case the value of that currency is at that moment lower than when he opened the position.

The last important concept before describing the intrinsic events to take into account is the **threshold**. The threshold is a certain percentage value, denoted by δ, which sets how much the market price should vary so that we consider that an intrinsic event has occurred. For example, for a threshold of 1%, if the market price that we take as reference is 5, an event will be considered when the price reaches either 4.95 or 5.05; see Figure 68

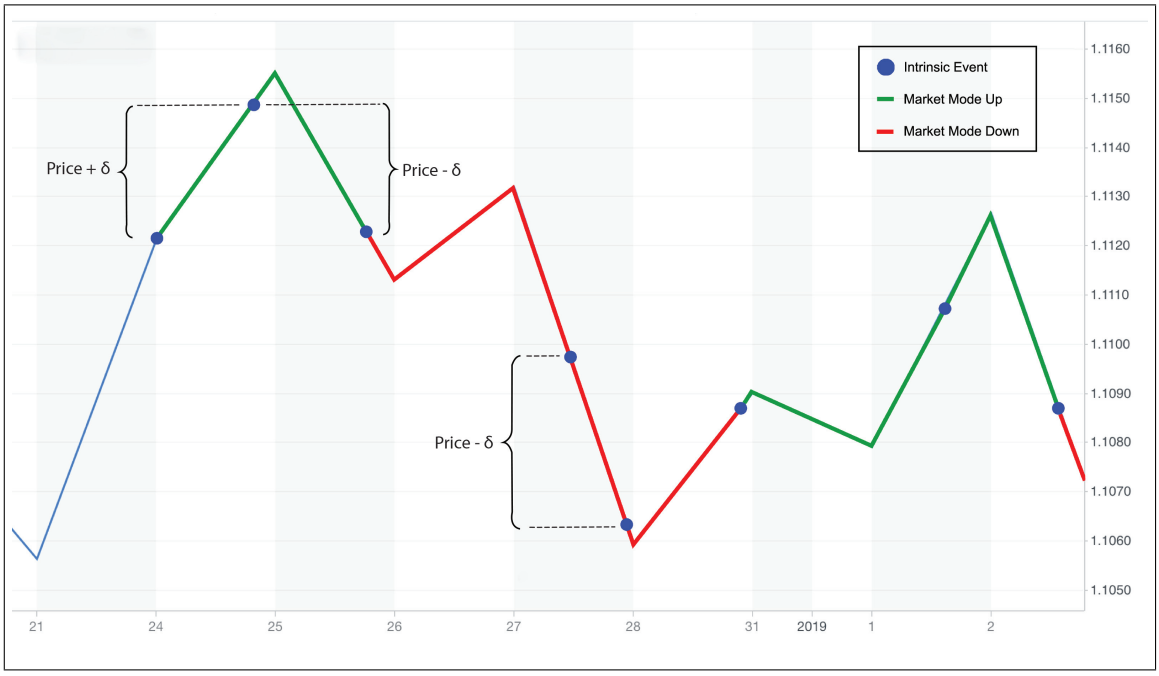


Figure 3.3: Threshold.

3.1.3 Intrinsic Events

Once all these previous concepts have been explained, we can talk about the events that will take place in the market based on the price, and, therefore, that our algorithm must detect.

Firstly, the event called directional-change, which will be given when an event occurs in the price series, i.e. the price varies by a $\delta\%$, and this event causes a change of market mode. This first event is denoted by α.

On the other hand, the event called *overshoot*, which is similar to directional-change, with the main difference that an overshoot does not mean a change of market mode. In other words, an overshoot is an event that follows the same market direction as the immediately previous event. The overshoot is going to be denoted by ω. See Figure 3.4 for an illustration of these concepts.

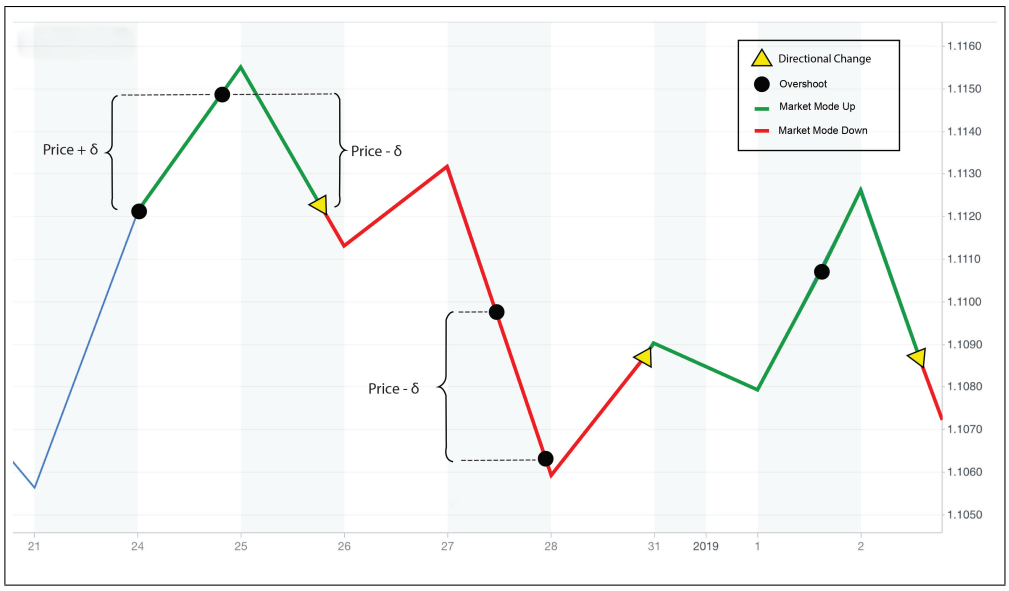


Figure 3.4: Intrinsic events.

To implement an algorithm capable to detect intrinsic events from a series of prices taken as input, we will design a function that is able to analyse time series and dissect them into α and ω events for a given $\delta\%$.

Nevertheless, before the design of this function, one last issue must be taken into account. The development of our algorithm is aimed for a real use, and not only for back-testing. Thus, in the Forex market, to buy or sell a currency we will find two different prices at a certain time: the purchase price, called **bid**; and the sale price, called **ask**.

The bid price is the price at which investors are willing to buy the currency., i.e., the bid price is the best price at which we can sell a currency at a certain time. Conversely, the ask price is the price at which investors are willing to sell, i.e., it is the best price at which we can buy a currency at a certain time.

Once these two concepts are properly understood, it is clear that the ask price will always

be greater than the bid price, but usually the difference between these two (called **spread**) is not very large, since these values are normally very similar to each other.

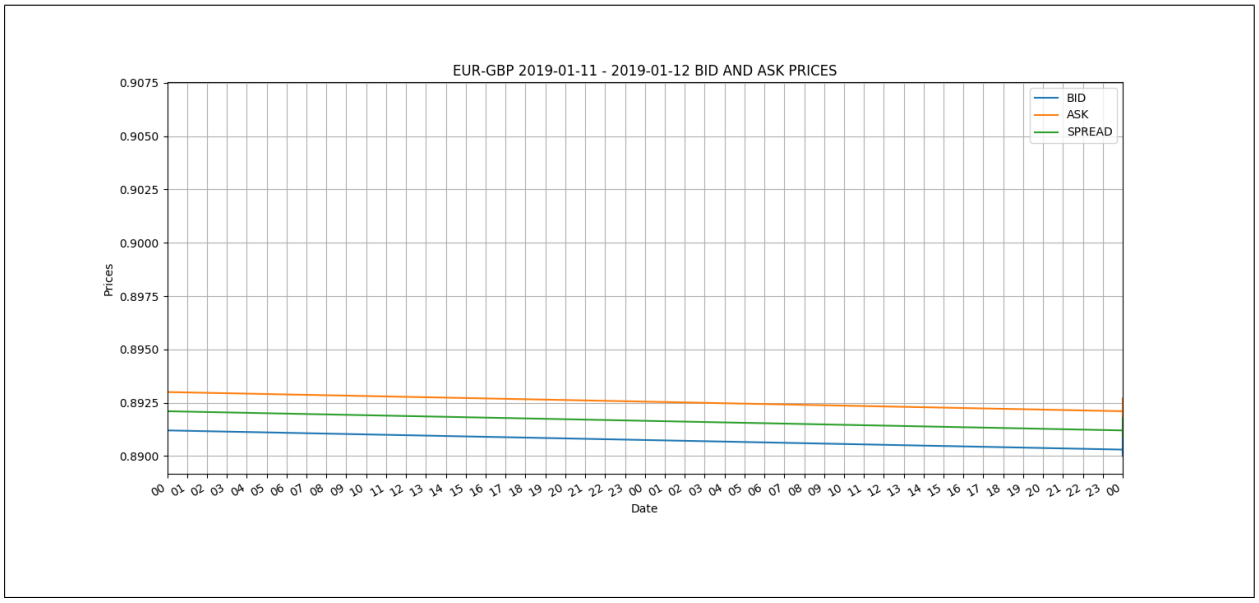


Figure 3.5: Bid and ask.

Returning to the topic of the algorithm design, we came up with an algorithm designed in 2017 by Golub, Glattfelder and Olsen called The Alpha Engine [22]. Specifically, The Alpha Engine is an automated trading algorithm based on the directional-change approach. The Alpha Engine code is public and can be consulted on GitHub [23].

Due to the similarity between the idea presented in The Alpha Engine and the algorithm that we want to develop, we are going to use the Alpha Engine's event detection model as a base. Thus, the function with which we will detect the directional-changes and overshoots is shown on next page.

Once the event detection function has been developed, the development of the algorithm has only just begun. The main question at this point is: once an event is detected, what does this event imply? This and other of the still unresolved aspects of the algorithm will be discussed in the next section.

3.2 Trading Strategy

3.2.1 The discovery of the Scaling Laws

Once it has been demonstrated that dissecting the price curve in an intrinsic-event time scale, done in a correct way, may suppose an important profit opportunity, the next point to be addressed is the trading strategy to follow when these events are detected.

The algorithm must be able to, given a directional change or an overshoot, decide if it is a good time to open (buy) or close (sell) an order. However, how can the algorithm effectively predict if after an event the market will follow one or another trend?

Algorithm 1 Algorithm that detects intrinsic event

```
1: function RECORD EVENT(PRICE)
2:
3:     if marketmode is up then
4:         if price.bid > extreme then
5:             extreme ← price.bid
6:             if price.bid > (reference + reference ∗ δ) then
7:                 reference ← extreme
8:                 return Up Overshoot
9:         else if price.ask < (reference − reference ∗ δ) then
10:            extreme ← price.ask
11:            reference ← price.ask
12:            marketmode ← down
13:            return Directional Change to Down
14:
15:    else if marketmode is down then
16:        if price.ask < extreme then
17:            extreme ← price.ask
18:            if price.ask < (reference − reference ∗ δ) then
19:                reference ← extreme
20:                return Down Overshoot
21:        else if price.bid > (reference + reference ∗ δ) then
22:            extreme ← price.bid
23:            reference ← price.bid
24:            marketmode ← up
25:            return Directional Change to Up
26:
27:    return No event
```

Facing this challenge, Glattfelder, Dupuis and Olsen in [24] performed a statistical analysis research on real time series of the Forex market based on the directional-change event approach, and as a result of their analysis, they discovered twelve new scaling laws that are applicable to this type of series.

According [25], in the context of statistics, a **scaling law** is a functional relationship between two quantities, where a relative change in one quantity results in a proportional relative change in the other quantity, independent of the initial size of those quantities: one quantity varies as a power of another. For instance, one well-known distribution that follows scaling laws is the **Pareto distribution**. The probability density function of a Pareto

random variable is

$$f(x) = \begin{cases} \alpha \dfrac{k^\alpha}{x^{\alpha+1}} & \text{if } x > k \\ \\ 0 & \text{otherwise} \end{cases}$$

where $\alpha, k > 0$ are real parameters, the former called the **shape parameter** and the latter **scale parameter**. Let X be a random variable with Pareto distribution of parameters α, k Ifi $m, x \in \mathbb{R}$. Furthermore, let x_1, x_2, a be real numbers such that $a > 0, x_1 > x_2$,and $x_2/a > k$. Then the following equality holds:

$$P(X > ax_1 \,|\, X > ax_2) = P(X > x_1/a \,|\, X > x_2/a)$$

This formula speaks itself of what a scaling law is. Scaling laws are also referred to as **scale-invariant laws**.

Scaling law relationships can be found in multiple natural processes, and they have been observed in different areas of knowledge such as physics, biology or computer science. Among the twelve new scaling laws discovered in [24], it is especially worth mentioning the following:

> On average, a directional-change event α of size $\delta\%$ is followed by an overshoot ω event of the same size.
>
> $$\langle \omega \rangle \approx \langle \alpha \rangle \tag{3.1}$$

This scaling law statistically suggests that once a directional change of a certain $\delta\%$ has been detected, it will be followed by an overshoot of the same magnitude of $\delta\%$.

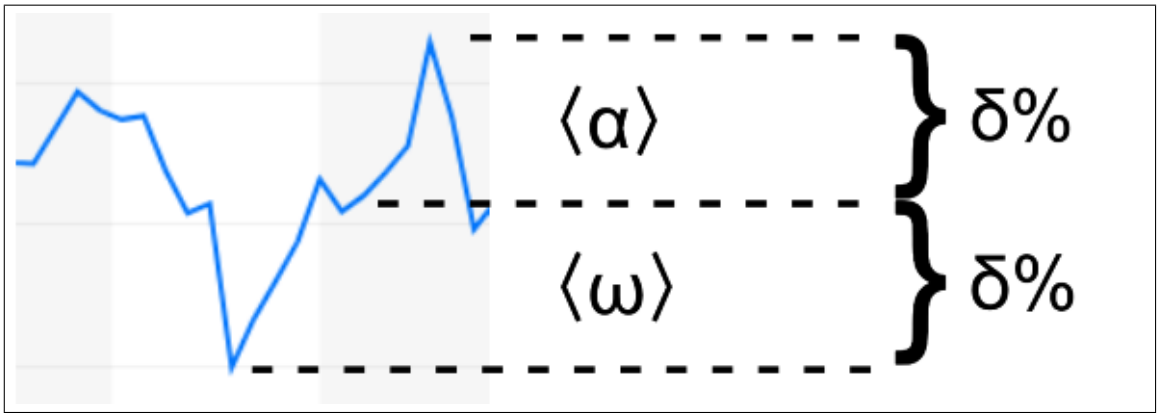


Figure 3.6: Scaling law for δ and ω

Derived from the discovery of this scaling law, Dupuis and Olsen in [26] introduced a new trading strategy that, by using the intrinsic events and knowing the properties of the aforementioned scaling law, can be applied to forecast market trends and decide when an order should be opened or closed. Dupuis and Olsen called this new trading strategy **coastline trader**.

3.2.2 The coastline trading strategy

Considering that, statistically and in average, a directional change is followed by an overshoot of the same length, the coastline trader strategy proposes to open a position whenever, according to the scaling law 3.1, the market is in a favourable state to our interests.

Going into detail, a coastline trader that invests with a **long trading strategy** will buy when an overshoot occurs and the market mode is down, or when a directional change occurs and the new market mode is up. On the other hand, a coastline trader investing with a **short trading strategy** will buy when an overshoot is detected and the market mode is up, or when a directional change is detected and the new market mode is down.

Once orders have been opened, coastline trader proposes to close them when an event is detected from which, following the statistics again, it is expected to have achieved the expected benefit for these positions.

In detail, a coastline trader algorithm that invests with a long trading strategy will close positions when an overshoot is detected with the market mode up and profit has been achieved in the open order: the current price of the currency is higher than in the moment the order was opened at. However, a coastline trader investing with a short trading strategy will close positions when an overshoot has been detected with the market mode down and a profit has been obtained for the opened order: the current price of the currency is lower than at the time the order was opened. See Figure 3.7 for an illustration of this discussion.

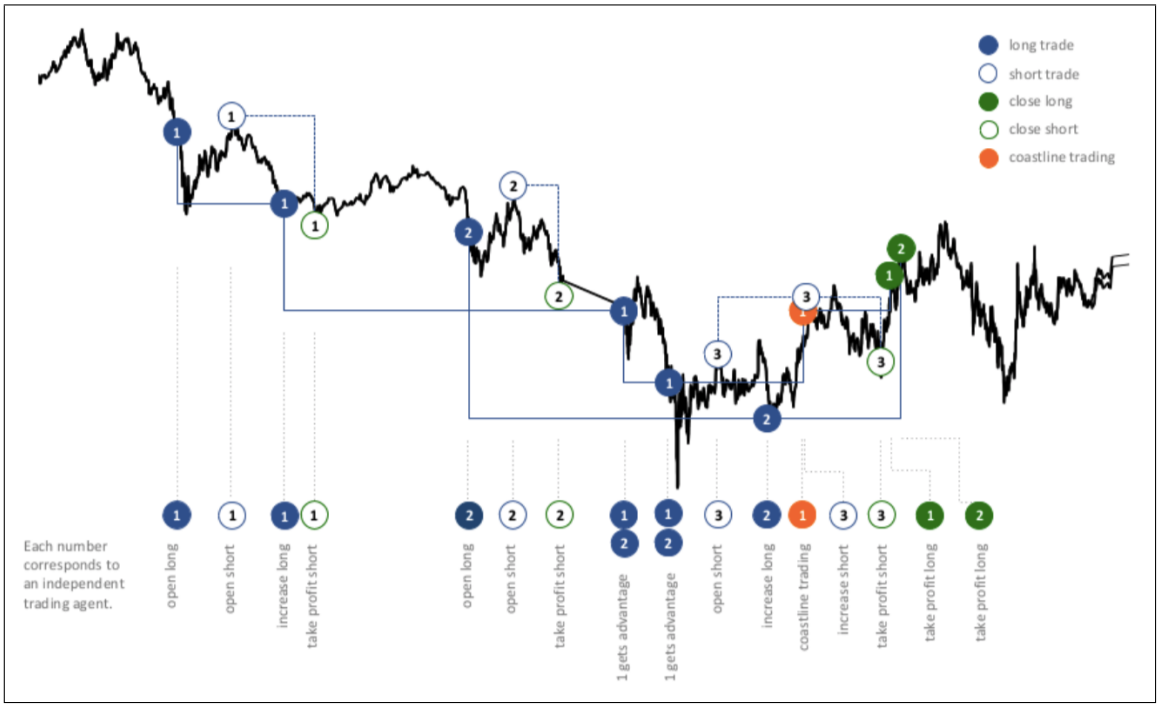


Figure 3.7: Coastline trading.

According to coastline trading strategy, the pseudocode of the function responsible for

executing operations in our algorithm, depending on the detected events is shown below. (Algorithm 2).

Algorithm 2 Algorithm that manages positions, depending on the detected event

```
1: function COASTLINE TRADING(EVENT)
2:
3:     if event is not 'No Event' then
4:
5:         if agentmode is long then
6:             if event is 'Down Overshoot' or 'Directional Change to Up' then return Buy
7:             else if event is 'Up Overshoot' and expectedProfit is True then return Sell
8:
9:         else if agentmode is short then
10:            if event is 'Up Overshoot' or 'Directional Change to Down' then return Buy
11:            else if event is 'Down Overshoot' and expectedProfit is True then return
    Sell
```

As stated in equation 3.1, due to the Scaling Law discovered, knowing we are going to open orders once a directional change has happened, we expect an overshoot of the same size to occur after. So that, the profit expected for an order will be of the same size of the directional change that triggered the buy.

The coastline trader model is a countertrend-trading model, which means that a price drop will cause the algorithm to buy, while a price increase will cause the algorithm to sell. This type of models provides liquidity to the market, since generally when the price of a currency moves down means that there is a lack of buyers for this currency, whereas if it moves up means that there are not enough sellers. This feature of the model is even beneficial to the markets as a whole.

The student of the University of Essex, Sanhanat Paniangtong, performed in his thesis [27] an analysis of various strategies derived from coastline trading using real data from various currencies and various indicators to measure how good and profitable each of the trading strategies would be. The conclusions of his analysis indicate that coastline trading can produce great benefits if certain elements were added to improve the strategy such as Trailing-Stop, aggressiveness control or a liquidity indicator. To be faithful to the truth, on its own, coastline trading is not a so effective system and can even produce losses in some situations. We will discuss this in deep detail in the conclusion section of this document.

Therefore, during the next sections, we will study different features that have been proposed by different researchers in order to achieve substantial improvement in the coastline trading strategy.

3.3 The Probability Indicator $\mathcal{L}$

3.3.1 The need of a market liquidity indicator

At this point, and after defining the different events that are going to be detected and the trading strategy to follow, we have an algorithm that is perfectly functional, that is, one capable of analyzing a series of prices taken as input and, based on these prices, if one of the possible intrinsic events occurs, and if applicable, open or close a market order. Nevertheless, the algorithm still has some weak points whose effects we will try to mitigate in this and the next sections.

One of these weaknesses of the algorithm is that, up to this point in the development, it is able to achieve benefits in a consistent manner only when the market follows a normal trend. On the other hand, when facing a strong market trend situation (sudden changes of price, high volatility, and the like), the algorithm opens many positions that in the long run will not be able to close with profit.

To illustrate this behaviour, we show below a table of the number of opened and closed orders, result of executing the algorithm using a series of prices of the currency market EUR / USD with a tick per minute from January 2001 to December 2019, simulating a long trading strategy agent with a threshold of 0.01%.

Year	Opened Orders	Closed Orders	Closed Percentage
2001	1425	1425	100
2002	869	869	100
2003	1080	1080	100
2004	1035	1035	100
2005	914	914	100
2006	610	610	100
2007	420	420	100
2008	2083	2029	97.4076
2009	1796	1784	99.3318
2010	1567	1567	100
2011	1537	1477	96.0963
2012	851	851	100
2013	590	590	100
2014	477	374	78.4067
2015	1419	1419	100
2016	892	892	100
2017	562	562	100
2018	703	619	88.0512
2019	104	76	73.0769

Table 3.1: Stats of the orders managed by the algorithm between 2001 and 2019.

As we can see in Table 5.5, from 2008, coinciding with the start of the Great Recession

[14], due to the strong market trend, the algorithm opens many orders that could not be closed even 11 year later, in 2019, i.e., the algorithm, at a certain moment, can reach to open many positions that in the long run will not be able to close with benefits. This factor causes more havoc in a short term than in a long term, because if one could wait indefinitely until closing the order with benefits, it would not be a problem for the system. The problem is generated because in the short term, and until the orders are closed with benefits, the capital used to open these positions is *seized*, so to speak, and will not be able to be used to open new orders. In order to solve this issue, Golub, Glattfelder and Olsen [22] added a new feature to the trading strategy of their Alpha Engine algorithm. This new feature is a probability indicator of the **liquidity** of the Forex market, based on the event-based price curve. This probability indicator is rooted in the extensive mathematical and statistical theory published in [28] and has the main goal of being able to anticipate and control how the algorithm will act depending on the different arriving market trends.

3.3.2 Insights from Information Theory

The probability indicator, denoted by $\mathcal{L}$, is a value of the information theory that measures the difference in the occurrence of price trajectories. What is actually analysed with this probability indicator is the evolution of the price mapped in the discretized price curve, whose results come from the event-based language, along with the overshoot scaling law.

In order to calculate this probability indicator, it is used the concept of point-wise **entropy**, or **surprise**, which theoretically is defined as the entropy of the realization of a random variable. According to [29], in statistical mechanics, entropy is an extensive property of a thermodynamic system. It is closely related to the number Ω of microscopic configurations (known as microstates) that are consistent with the macroscopic quantities that characterize the system (such as its volume, pressure and temperature). Under the assumption that each microstate is equally probable, the entropy S is the natural logarithm of the number of microstates (multiplied by a constant). The concept of entropy is also used in information theory. In all cases, entropy is conceived as a"measure of disorder" or the "peculiarity of certain combinations". Entropy can be considered as a measure of the uncertainty and information necessary to, in any process, limit, reduce or eliminate uncertainty. Information entropy is the average rate at which information is produced by a stochastic source of data.

The measure of information entropy associated with each possible data value is the negative logarithm of the probability mass function for the value:

$$S = -\sum_i P_i \log P_i$$

where P_i is the probability mass function.

When the data source produces a low-probability value (i.e., when a low-probability event occurs), the event carries more "information" ("surprisal") than when the source data produces a high-probability value. The amount of information conveyed by each event defined in this way becomes a random variable whose expected value is the information entropy.

To apply these concepts of information theory to the event-based curve of Forex prices, following [30], it is understood that the surprise of the event-based price curve will be related to the probabilities of transitions from the current state, denoted by s_i, to the next intrinsic event, denoted by s_j, i.e., $\mathbb{P}(s_i \to s_j)$. Since the only intrinsic events that we will take into account in the algorithm are the directional changes δ and the overshoots ω, we know that the **transition network** of states in the event-based representation of the price trajectories is represented by the following diagram.

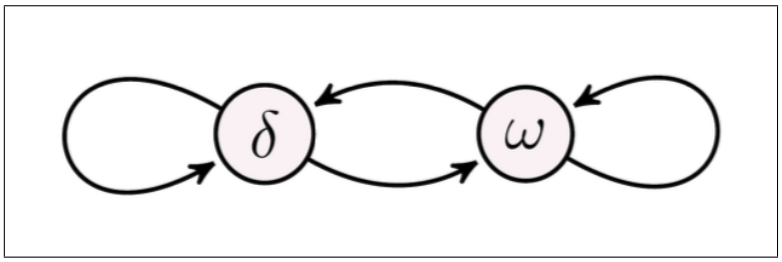


Figure 3.8: The transition network of states in the directional-change approach representation of the price trajectories ω [22]

By looking at Figure 3.8, we know that the possible evolution of states from a directional change can be either another directional change or an overshoot; and, similarly, the possible evolution of states from an overshoot can be either another overshoot or a directional change. Keeping this in mind, in [22], the surprise of transitions from a state s_i to a state s_j is defined as:

$$\gamma_{i,j} = -\log \mathbb{P}(s_i \to s_j) \tag{3.2}$$

If the transition from state s_i to s_j is very unlikely, the value of $\gamma_{i,j}$ calculated using the previous equation will be very large.

$$\begin{cases} \mathbb{P}(s_i \to s_j) \approx 0, \text{ then } \gamma_{i,j} \gg 0 \\ \mathbb{P}(s_i \to s_j) \approx 1, \text{ then } \gamma_{i,j} \approx 0 \end{cases} \tag{3.3}$$

Consequently, and from Equation 3.1, it can be derived the surprise of a series of prices in a time interval [0, T] in which K transitions from a state to another one have been experienced, such as the sum of the surprises of each one of the K transitions. Thus, the resulting equation, called the **surprise equation** is:

$$\gamma_K^{[0,T]} = \sum_{k=1}^{K} -\log \mathbb{P}(s_{i_k} \to s_{i_{k+1}}) \tag{3.4}$$

This equation can be used now as a measure of the the unlikeliness of a price path. It is worth mentioning that it is a price trajectory dependent measure. In other words, two different price trajectories that exhibit the same volatility may have very different surprise

values. In case that the value of the surprise of the price trajectory is large, it will indicate that the price trajectory has experienced an unexpected movement.

Following, again, the article [28], and assuming that we want to be able to calculate the surprise of a price trajectory of a certain time interval [0, T] using real price data, applying Equation 3.4 we are going to encounter a major problem: it may happen that in some periods of time the algorithm may detect very few transitions of intrinsic states, while in periods of greater activity the algorithm will detect many more transitions of states, which may affect the fidelity of the value of the resulting surprise. In order to remove the market activity of surprise calculations, we need to eliminate the component related to the number of transitions from the surprise within the aforementioned time interval.

Using the Shannon-McMillan-Brieman theorem on convergence of **sample entropy** [31] and the corresponding central limit theorem [32], they refer to important quantities, which will be used in the final expression of the liquidity indicator. The first is $H^{(1)}$ and denotes the **entropy rate** associated with the state transitions.

$$H^{(1)} = \sum_{i=0}^{2^n-1} \mu_i \mathbb{E}[-\log \mathbb{P}(s_i \to \cdot)] \tag{3.5}$$

where μ is the stationary distribution of the corresponding Markov chain. The authors made a strong hypothesis in order to simplify the equations and that was assuming that the state at a given time point only depends on the previous time point. In a previous paper, the authors claimed that even though they don't have a formal proof of this fact, the results they obtained by making this hypothesis were good enough to keep it. That is why we see the μ_i values in the equation above.

The next equation to be taken into account is the following.

$$H^{(2)} = \sum_{i=0}^{2^n-1} \sum_{j=0}^{2^n-1} \mu_i \mu_j Cov\left(-\log \mathbb{P}(s_i \to \cdot), -\log \mathbb{P}(s_j \to \cdot)\right) \tag{3.6}$$

$H^{(2)}$, is the **second order of informativeness**.

These two equations are used to redefine the surprise of a price trajectory, centered on its expected value as:

$$\Delta = \frac{\gamma_K^{[0,T]} - K \cdot H^{(1)}}{\sqrt{K \cdot H^{(2)}}} \tag{3.7}$$

Following the central limit theorem [33], the centered expression of the surprise of a price trajectory converges to the Normal Distribution when the number of transitions K tends to infinity:

$$\frac{\gamma_K^{[0,T]} - K \cdot H^{(1)}}{\sqrt{K \cdot H^{(2)}}} \to N(0,1) \text{ for } K \to \infty \tag{3.8}$$

Thus, we can finally define the **liquidity probability indicator** with as following equation:

$$\mathcal{L} = 1 - \Theta\left(\frac{\gamma_K^{[0,T]} - K \cdot H^{(1)}}{\sqrt{K \cdot H^{(2)}}}\right) \tag{3.9}$$

where Θ is the cumulative distribution function of a standard normal distribution.

When the market shows unpredictable behaviour, the probability indicator $\mathcal{L}$ will have an approximate value of 0. Therefore, we can quantify how normal is the market behaviour and act accordingly: the market shows normal behavior when the value of $\mathcal{L}$ is approximate 1.

3.3.3 The implementation of $\mathcal{L}$

Finally, for this probability indicator to be efficient, we will have to redefine the way the overshoot event ω should be detected. As explained in Section 3.1 (intrinsic time), an overshoot occurs when the price changes a threshold percent in the direction of the market mode. In the context of the coastline trading event detection, this overshoot definition will not change, but, nevertheless, following [28] calculations, in the context of the probability indicator, for the events that we will use to calculate the value of $\mathcal{L}$, it will be considered an overshoot only when the price varies a $2.525729 \cdot \delta\%$. The value 2.525729 comes from maximizing the second order informativeness $H^{(2)}$. Indeed, the following inequality holds for the **sample entropy** $H_n^{(2)}$

$$H_n^{(2)} \leq \max_{s_i \in \mathcal{S}} \left\{ \mathrm{Var}(-\log P(s_i \to \cdot))^2 \right\}^2 \tag{3.10}$$

where $\mathcal{S}$ is the set of states built during the discretization process of the price curve. By the central limit theorem we know that for large n $H_n^{(2)}$ tends to $H^{(2)}$. By taking the maximum value in the right part of this inequality, we obtain the value 2.525729; see [34], [22] for more technical details.

As for $H^{(1)}$, there is another upper bound that will allow us to enormously simplify the calculations. Such upper bound is

$$H_n^{(1)} \leq \max_{s_i \in \mathcal{S}} \{ \mathrm{E}(-\log P(s_i \to \cdot)) \} \tag{3.11}$$

The same reasoning applies for $H^{(1)}$; by the central limit theorem, we know that $H_n^{(1)} \to H^{(1)}$ as n tends to infinity.

To sum up, in order to calculate $\mathcal{L}$ value, we will develop a new event detector, setting the threshold value to $2.525729 \cdot \delta\%$ for the overshoot events (Algorithm 3):

Algorithm 3 Algorithm that detects intrinsic event

```
1: function RECORD LIQUIDITY EVENT(PRICE)
2:
3:     if marketmode is up then
4:         if price.bid > extreme then
5:             extreme ← price.bid
6:             if price.bid > (reference + reference ∗ δ) then
7:                 reference ← extreme
8:                 return Up Overshoot
9:         else if price.ask < (reference − reference ∗ δ) then
10:            extreme ← price.ask
11:            reference ← price.ask
12:            marketmode ← down
13:            return Directional Change to Down
14:
15:    else if marketmode is down then
16:        if price.ask < extreme then
17:            extreme ← price.ask
18:            if price.ask < (reference − reference ∗ δ) then
19:                reference ← extreme
20:                return Down Overshoot
21:        else if price.bid > (reference + reference ∗ δ) then
22:            extreme ← price.bid
23:            reference ← price.bid
24:            marketmode ← up
25:            return Directional Change to Up
26:
27:    return No event
```

Each time a new event is detected, the value of $\mathcal{L}$ will be updated using the equations $H^{(1)}$, $H^{(2)}$, Surprise and $\mathcal{L}$ as shown in the following algorithm:

Algorithm 4 Algorithm that updates the value of $\mathcal{L}$ every time a new liquidity intrinsic event is detected

```
1: function UPDATE L(EVENT)
2:
3:     if event is not 'No Event' then
4:
5:         if event is 'Directional Change' then
6:             surprise = weight * 0.08338161 + (1 - weight) * surprise
7:
8:         else if event is 'Overshoot' then
9:             surprise = weight * 2.52579 + (1 - weight) * surprise
10:
11:        L = 1 - NormalDistributionCumulative (sqrt(K) * (surprise - H(1)) / H(2) )
12:        return L
```

Being the variable *weight* a fixed value used to compute the mean of the surprise over K transactions and the function NormalDistributionCumulative a method that returns the cumulative distribution function of the standard normal distribution, previously denoted by Θ. So that, after getting an event distinct of 'No Event', the algorithm first computes the new value of the surprise and then uses this new value to compute the current liquidity $\mathcal{L}$.

By using the probability indicator $\mathcal{L}$, we can control the aggressiveness of the algorithm, i.e., the capital amount we will use to open an order, whereby, a better risk-adjusted performance will be guaranteed.

In detail, following again the AlphaEngine model, when a position is to be opened, the unit size (volume of capital we will open the order with) will be reduced when the $\mathcal{L}$ indicator starts to approach to 0 of the next shape:

- If $\mathcal{L} < 0.5$ unit size is reduced by 50%
- If $\mathcal{L} < 0.1$ unit size is reduced by 90%

The pseudocode of this function is shown bellow:

Algorithm 5 Algorithm that updates the value of the unit size depending on the current value of $\mathcal{L}$

```
1: function GET UNIT SIZE (L)
2:
3:     if L ≤ 1 and L > 0.5 then return OriginalUnitSize
4:
5:     else if L ≤ 0.5 and L > 0.1 then return OriginalUnitSize * 0.5
6:
7:     else if L ≤ 0.1 then return OriginalUnitSize * 0.1
```

The implementation of the aforementioned features allows the trading algorithm to work better in situations that it could not control previously. However, at this point there are

still missing factors that can and should be added to the algorithm so that it works up to its full potential, which we will explain in future sections.

3.4 Asymmetric thresholds

3.4.1 The last feature of the Alpha Engine

In the previous section, we accomplished the development of a probability indicator that is able to show us with confidence the liquidity of the market, and, by using this value, we will be able to control the aggressiveness with which our algorithm will open positions. In this section, we are going to follow again the path in The Alpha Engine [22] and try to implement successfully the last feature of this algorithm, which significantly differentiates and improves their trading strategy when compared to a traditional coastline trader.

The last feature that researchers Golub, Glattfelder and Olsen added to The Alpha Engine was allowing new degrees of freedom to the trading model. This is the last piece of the puzzle with which they concluded the development of their algorithm, and from which they began the back-testing phase using historical data of different currencies. The results of these executions are shown in Figure 3.9.

From the Figure 3.9, several conclusions are drawn:

- The first and most clear conclusion is that, once all the functionalities have been developed, a great daily benefit can be obtained by using The Alpha Engine algorithm in Forex market. Regarding this first conclusion, it is a justification to implement all the functionalities of The Alpha Engine in our algorithm.
- The second conclusion is that the algorithm will not have the same effectiveness for all the pairs of currencies, and even for some of these it will result in losses. This second conclusion is going to entail the need to carry out an analysis, through tests with historical data, of which pairs of currencies will be appropriate to use our algorithm on and in which pairs of currencies it will be better not to trade. This issue will be discussed later, once the functionalities are added to our algorithm and we can proceed to the testing part.

To sum up, by observing Figure 3.9, there is a justification to implement the latest novel functionality that includes The Alpha Engine. Going into detail, this functionality allows the trading model to have more freedom, converting what previously were static thresholds into dynamic thresholds that adapt their value according to the market situation, and its development will be discussed in the next subsection.

3.4.2 Introducing δ_{up} and δ_{down}

The objective of the development that we will carry out this section is to vary in a suitable way the value of $\delta\%$ used to detect intrinsic events, according to the trend that the market

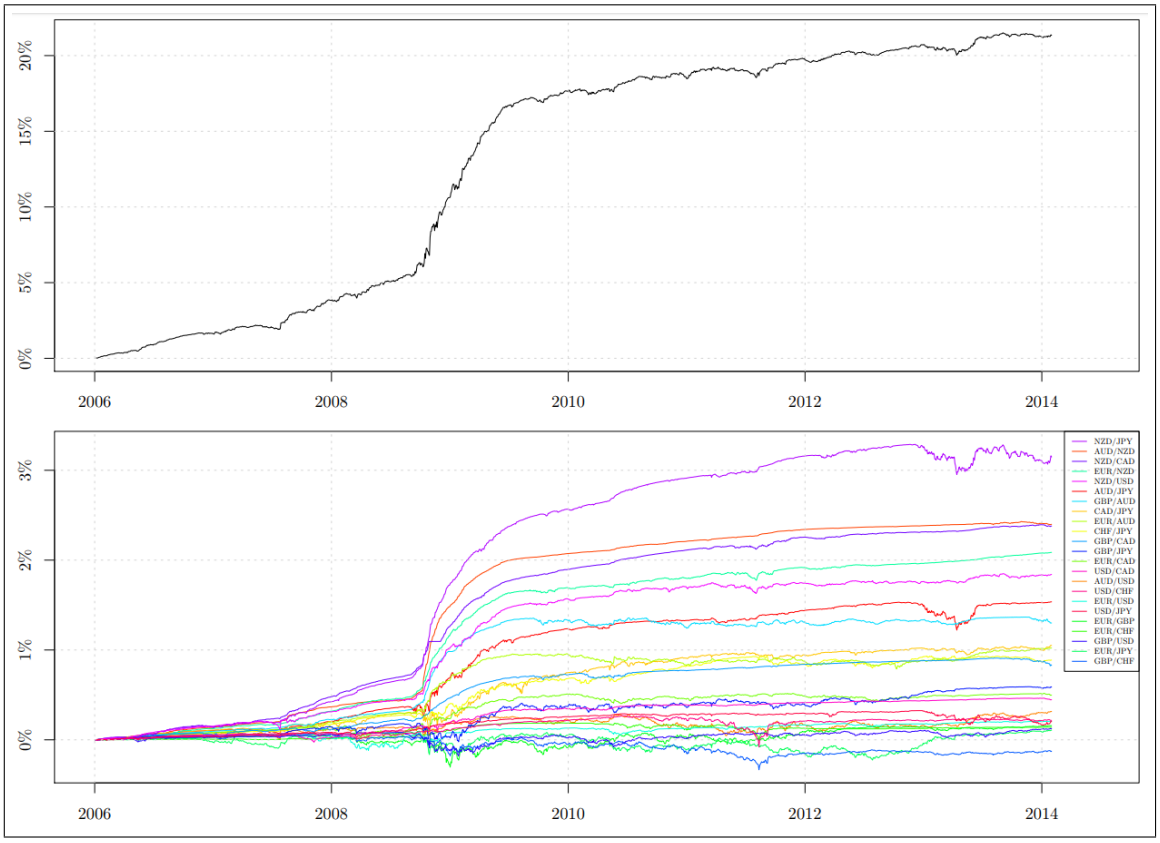


Figure 3.9: Daily Profit & Loss of the Alpha Engine, across 23 currency pairs, for eight years. [22]

is following at a certain moment. It should be noted that $\delta\%$ refers to the threshold value. When a price that we take as a reference varies by a certain $\delta\%$ we will consider that an intrinsic event has occurred in the market price series. Thus, it is reasonable that the value $\delta\%$ is dynamic and that the algorithm automatically modifies it according to the market trend. For instance, when the algorithm encounters a very strong up trend, the threshold with which the overshoot for the market mode is detected should be increasingly becoming smaller and smaller, with the aim that the stronger the trend, the more intrinsic events can detect the algorithm, and therefore more operations can be executed.

An important question to answer arises at this point. Knowing that the variations in the value of the threshold depends on the market trend, does it make sense to take into account two different threshold values, δ_{up} and δ_{down}, that serve to detect events according to whether the direction of the movement of the price is respectively up and down?

Golub, Glattfelder, Petrov and Olsen in [35], perform multiple Monte Carlo simulations of the **number of directional changes** N detected using asymmetric directional change thresholds δ_{up} and δ_{down}. Specifically, in this study they carry out simulations for the cases of no trend, positive trend and negative trend of the market. In Figure 3.9 it is shown the results of these Monte Carlo simulations. By analysing these results, we reach several

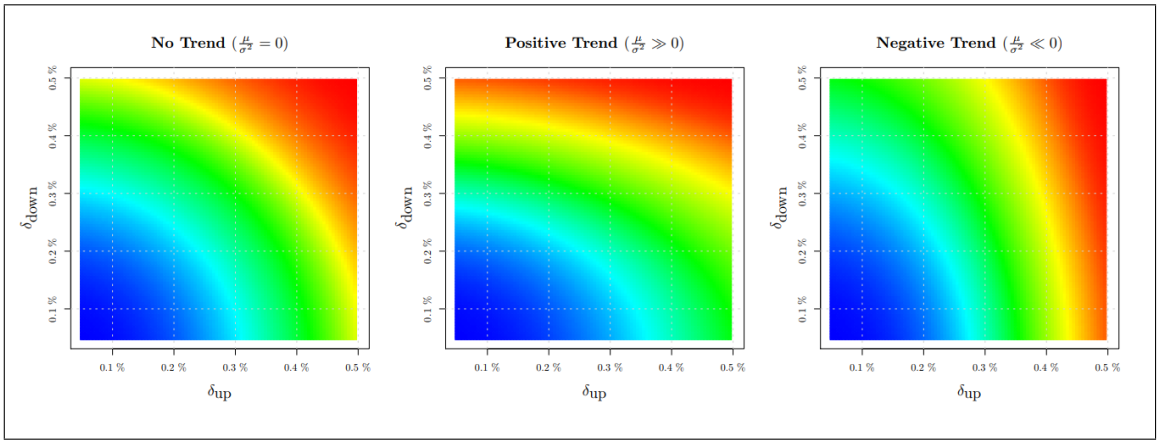


Figure 3.10: Monte Carlo simulation of the number of directional changes N as a function of the asymmetric directional change thresholds δ_{up} and δ_{down}. [22]

conclusions:

- For the no market trend situation, we observe that the contour lines are perfect circles. This shows that the same number of directional changes are found for the corresponding asymmetric thresholds δ_{up} and δ_{down}. Therefore, in case of no trend, there is no difference between using a single threshold or two asymmetric thresholds.
- For the positive market trend situation, we observe that, comparing with the result of the no market trend simulation, the countour lines shifts to the right. This means that the threshold δ_{up} improves the number of directional changes detected using a single threshold, as long as δ_{up} is greater than δ_{down}.
- In contrast to the positive market trend situation, for the negative market trend, we can observe that, the countour lines shifts to the left. This means that the threshold δ_{down} improves the number of directional changes detected using a single threshold, as long as δ_{down} is greater than δ_{up}.

By virtue of the results and conclusions obtained from Monte Carlo simulations, there is an empirical reason to, from this point, start using two asymmetric thresholds to detect events depending on the market mode. Thereby, the new thresholds will be defined by:

$$\begin{cases} \delta_{up} \text{ for } \alpha_{down} \text{ and } \omega_{up} \\ \delta_{down} \text{ for } \alpha_{up} \text{ and } \omega_{down} \end{cases} \tag{3.12}$$

These new thresholds will henceforth record intrinsic events with different percentage variations of the price taken as reference, depending on the direction taken by the movement of the price.

This change in the thresholds will cause the code of the intrinsic event detection functions described in Subsections 3.1.3 and 3.3.3 to be changed as well at this point. The new

pseudocode of the intrinsic event detection function is shown in the Algorithm 5 shown in the next page.

Algorithm 6 Algorithm that detects intrinsic event using asymmetric thresholds

```
1: function RECORD EVENT(PRICE)
2:
3:     if marketmode is up then
4:         if price.bid > extreme then
5:             extreme ← price.bid
6:             if price.bid > (reference + reference * δ_up) then
7:                 reference ← extreme
8:                 return Up Overshoot
9:         else if price.ask < (reference − reference * δ_down) then
10:            extreme ← price.ask
11:            reference ← price.ask
12:            marketmode ← down
13:            return Directional Change to Down
14:
15:    else if marketmode is down then
16:        if price.ask < extreme then
17:            extreme ← price.ask
18:            if price.ask < (reference − reference * δ_down) then
19:                reference ← extreme
20:                return Down Overshoot
21:        else if price.bid > (reference + reference * δ_up) then
22:            extreme ← price.bid
23:            reference ← price.bid
24:            marketmode ← up
25:            return Directional Change to Up
26:
27:    return No event
```

The changes respect to the Algorithm 1 defined in Subsection 3.1.3, made with the aim of introducing the new thresholds δ_{up} and δ_{down}, must be also applied in the function of detection of intrinsic events for the liquidity indicator, i.e. for the Algorithm 3 defined in Subsection 3.1.3. Recall that the difference between algorithms 1 and 3 is that, in the case of the detection of intrinsic events for the computation of the liquidity indicator, the thresholds must be multiplied by the value 2.525729. A more detailed explanation can be found in Section 3.3.

Once the new asymmetric thresholds have been introduced in the event detection, [22] proposes a new way to use these thresholds to improve the functioning of the algortithm. In particular, researchers seek a way of, knowing the trend that is following the market at a certain moment, screw the thresholds accordingly to compensate.

Getting to vary the value of the thresholds δ_{up} and δ_{down} depending the market trend, allow us to detect more events to close the orders that we have previously opened. In particular, the operation that should follow this variation in the values of the thresholds is the following:

$$\begin{cases} \text{When no market trend, then } \delta_{up} = \delta_{down} \\ \text{When market trend is up, then } \delta_{up} < \delta_{down} \\ \text{When market trend is down, then } \delta_{down} > \delta_{up} \end{cases} \tag{3.13}$$

However, once again a big problem must be faced in the implementation of this feature: given a certain price series, how does the algorithm determine if the market follows a non-trend situation, a positive trend situation or a negative trend situation?

With this aim in mind, [22] propose using the concept of **inventory** as a proxy or gauge for the market trend. The inventory of a trader indicates the amount of currencies that, at a certain moment, the trader owns among all his open orders. For instance, given a coastline trader that is investing in a certain market such as the EUR / USD, and has opened without being able to close the following orders: a long order with volume 4, five long orders with volume 2 and three short orders with volume 1; The value of the inventory of the trader, denoted by $\mathcal{I}$ will be 11, result of the following operation:

$$\mathcal{I} = 1 \cdot 4 + 5 \cdot 2 - 3 \cdot 1 = 11 \tag{3.14}$$

Therefore, the inventory can be defined as:

$$\mathcal{I} = (1 \cdot \sum \text{Long Order Volume}) + (-1 \cdot \sum \text{Short Order Volume}) \tag{3.15}$$

where the volume of an order is given by the amount of currencies purchased when the order was opened (for more details, see the *Unit Size* concept in the subsection 3.3.3)

The function that computes the value of the inventory using all the opened orders, and that must be called every time an order is opened or close is shown in the following page (Algorithm 7).

3.4.3 Using the inventory to gauge the market trend

Once the concept of inventory is explained, the solution proposed in [22] is to use the value of a trader's inventory to measure the market trend.

In the first place, if the inventory of a coastline trader is a very high value with a positive or negative sign, it means that the trader has opened many long or short orders that could not have been closed at that moment. If this happens, the algorithm should reduce the value of the thresholds in order to detect more intrinsic events to close those orders easily. Recall that in Section 3.2 we define that the long orders will be closed when an overshoot happens while up market mode is detected. Therefore, it makes sense to reduce the value of the threshold δ_{up} when the inventory is a very high value, with the objective that the algorithm detects a larger number of up overshoots that trigger a sell of the orders.

Algorithm 7 Algorithm that updates the value of $\mathcal{I}$ depending on the volume of the opened orders

```
1: function Get Inventory (Orders)
2:
3:     I = 0
4:
5:     for order in Orders do
6:
7:         if order.agentMode is long then
8:             I = I + textitorder.unitSize
9:
10:        else if order.agentMode is short then
11:            I = I - textitorder.unitSize
12:
13:    return I
```

Conversely, if the inventory of a coastline trader is a very high value, but with negative sign, it means that the trader has opened many short orders that could not have been closed at that moment, so the trader should screw the value of thresholds in order to close that orders. In this case, the short orders will be closed when an overshoot happens while down market mode is detected. Therefore, it makes sense to reduce the value of the threshold δ_{down} when the inventory is a very high negative value, with the objective that the algorithm detects a large number of down overshoots.

The implementation of the solution proposed with which the value of the thresholds will be adjusted according to the value of inventory is given by the following equation, corresponding to a long position:

$$\frac{\delta_{down}}{\delta_{up}} = \begin{cases} 2 \text{ if } \mathcal{I} \geq 15; \\ 4 \text{ if } \mathcal{I} \geq 30. \end{cases} \tag{3.16}$$

And the same equation, but inverted, for a short position:

$$\frac{\delta_{up}}{\delta_{down}} = \begin{cases} 2 \text{ if } \mathcal{I} \leq -15; \\ 4 \text{ if } \mathcal{I} \leq -30. \end{cases} \tag{3.17}$$

From these two previous equations, the variation of the thresholds values will be given for the next equation:

$$\begin{cases} \text{If } \mathcal{I} \geq 15, \text{ then } \delta_{up} = 0.75 \cdot \delta_{up} \text{ and } \delta_{down} = 1.5 \cdot \delta_{down} \\ \text{If } \mathcal{I} \geq 30, \text{ then } \delta_{up} = 0.5 \cdot \delta_{up} \text{ and } \delta_{down} = 2 \cdot \delta_{down} \\ \text{If } \mathcal{I} \leq -15, \text{ then } \delta_{up} = 1.5 \cdot \delta_{up} \text{ and } \delta_{down} = 0.75 \cdot \delta_{down} \\ \text{If } \mathcal{I} \leq -30, \text{ then } \delta_{up} = 2 \cdot \delta_{up} \text{ and } \delta_{down} = 0.5 \cdot \delta_{down} \end{cases} \tag{3.18}$$

The function that allows us to make these fractional changes in the value of the thresholds, and that will be implemented in our algorithm is described in the Algorithm 8 below.

Algorithm 8 Algorithm that adjusts the value of the asymmetric thresholds depending on the value of $\mathcal{I}$

function ADJUST THRESHOLDS($\mathcal{I}$)

    if $\mathcal{I} \geq 150$ and $\mathcal{I} \leq 299$ **then**
        $\delta_{up} = \delta_{original} * 0.75$
        $\delta_{down} = \delta_{original} * 1.5$

    else if $\mathcal{I} \geq 300$ **then**
        $\delta_{up} = \delta_{original} * 0.5$
        $\delta_{down} = \delta_{original} * 2$

    else if $\mathcal{I} \geq -299$ and $\mathcal{I} \leq -150$ **then**
        $\delta_{up} = \delta_{original} * 1.5$
        $\delta_{down} = \delta_{original} * 0.75$

    else if $\mathcal{I} \leq -300$ **then**
        $\delta_{up} = \delta_{original} * 2$
        $\delta_{down} = \delta_{original} * 0.5$

This function, which updates the values of the thresholds δ_{up} and δ_{down} depending of the value of inventory, must be called each time a trading action occurs, i.e., every time a long or short order is opened or closed.

In the following figure, we can observe the different behaviour of the event detection algorithm after applying this last functionality.

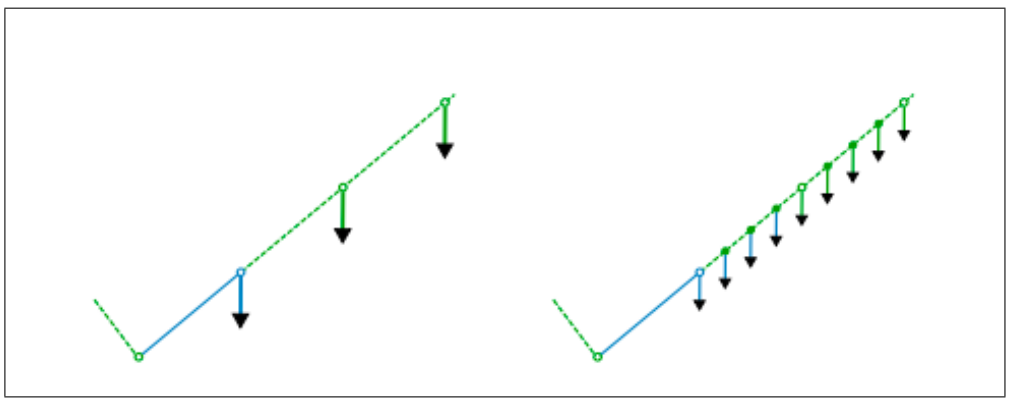


Figure 3.11: Different behaviour of the intrinsic event detection using fixed (left) and asymmetric (right) thresholds. [22]

Once the new asymmetric thresholds δ_{up} and δ_{down} have been added, the inventory concepts has been introduced and the thresholds value has been screwed according the inventory value to get a better performance for our algorithm, all the features that Golub, Glattfelder

y Olsen introduced in The Alpha Engine [22] are fully integrated in our algorithm.

At this point, the algorithm is completely functional and, as discussed in the subsection 3.4.1, offers very positive results. Nevertheless, as the authors of The Alpha Engine themselves state in their paper, the algorithm should be understood as a prototype that, after being tested, shows that directional-change approach will be able to achieve great results and advances in the high-frequency algorithmic trading in a future, but at the moment is far from being firmly completed and implemented. In particular, they note that there are several possible features to add to the algorithm with which from this point the development could be continued.

Among the possible features to be implemented in the algorithm The Alpha Engine, it is mentioned that could be added a function that calibrates the various exchange rates by volatility, or by excluding illiquid ones. It could be also added correlation across currency pairs in order to increase the performance of the trading model. Finally, it is stated that a whole layer of risk management could be added for the model.

Our novelty and contribution to The Alpha Engine algorithm will go through, precisely, developing a risk control system for open orders that allows us to close certain orders when it is considered preferable to assume a loss before it becomes worse. This functionality will be the last one that we will add to our algorithm before finishing with its development and beginning with the testing phase. The details of this latest development will be presented in the next section.

3.5 The Risk Management Layer

3.5.1 The importance of managing the risk

As it has been mentioned at the end of the previous subsection, our novel contribution and last characteristic to add to the high-frequency trading algorithm will be a risk control system of the actions that are opened and have not been able to close after a given time period. The objective behind this risk control system is, given a new market price, to determine which of the opened orders that could not be closed with profit at that moment are we interested in holding, expecting them to obtain benefits in the short or medium term, and which we want to get rid off before the loss is greater.

In none of the documents taken as reference for the development of the algorithm, up to this point, has been mentioned any risk control strategy that could be effective or that has been implemented with positive results. So that, for the implementation of this feature, we will have to do a research work to determine what is the optimal strategy and if it finally is worth, or not, to be implemented in the final version.

Although in the development of The Alpha Engine [22] the authors do not include any Risk Management strategy, as it can interfere with the correct functioning of the algorithm, it is worth noting that there is a great danger if we let an order remain opened for an undetermined time. Indeed, the importance of risk management in trading is huge, since often managing risk also means protecting your capital. If no Risk Control System is implemented,

then the money invested to open an order could remain blocked for an indefinite time. So that, in certain situations, since every trader makes losing trades sometimes, it is better to assume the loss and recover part of the investment and use this capital to reinvest in new shares that, hopefully, will have a better result.

Currently there are multiple risk management techniques used by traders such as One-Percent Rule, Stop-Loss, Take-Profit, Risk-Reward Ratio or Trailing-Stop. None of these techniques has been successfully implemented so far in an algorithm that uses the Coastline Trading strategy, so we will have to study all the possibilities and determine which one is the optimum to be integrated in our system.

3.5.2 Stop-Loss

Firstly, one of the simplest and most familiar risk management techniques will be explained bellow: the **Stop-Loss**. Stop-Loss orders are shares opened in a broker, which upon reaching a certain price will be closed. The main objective of Stop-Loss is to set a value, usually a percentage, with which to limit the losses of a trader for a position. In this way, when the trader opens an order, long or short, a price limit will be automatically set. Once this price is reached, it will cause the order to close even though a profit has not been obtained [36].

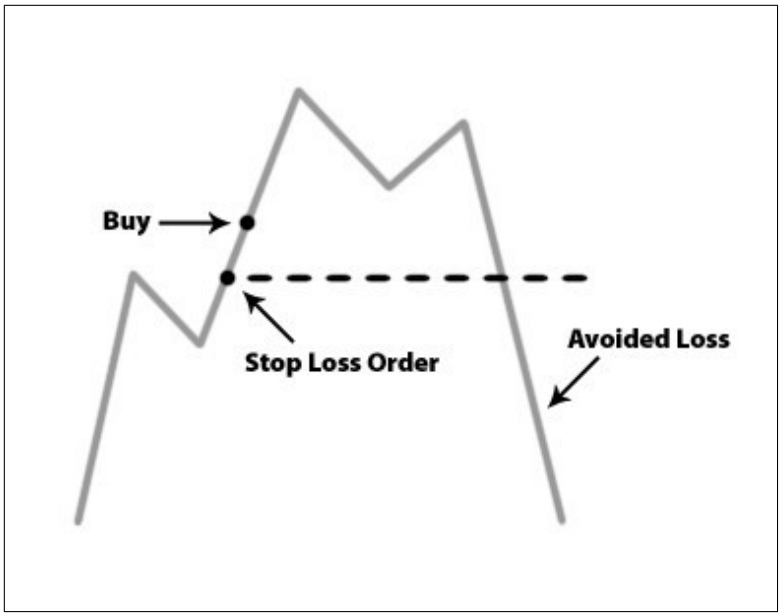


Figure 3.12: Stop Loss

For instance, given a trader in the EUR/USD market opens a long order at a price of 1.1209 and sets a Stop-Loss of 2%, if the market price reaches 1.0984, the trader will determine that the loss should be assumed and close the order.

The function that computes the price of Stop-Loss, and that must be called each time an order is opened is the following:

Algorithm 9 Algorithm that sets the Stop-Loss price of an order

```
1: function Set StopLoss(Order, StopLoss)
2:
3:     if order.agentMode is long then
4:         order.stopLoss = currentPrice * (1 - stopLoss)
5:
6:     else if order.agentMode is short then
7:         order.stopLoss = currentPrice * (1 + stopLoss)
```

Meanwhile the function of a Stop-Loss algorithm for a series of prices would be as follows:

Algorithm 10 Stop-Loss Algorithm

```
1: function StopLoss()
2:
3:     for each new marketPrice do :
4:
5:         for each order in openedOrders do :
6:
7:             if order.agentMode is long and order.stopLoss > marketPrice then
8:                 sellOrder()
9:             else if order.agentMode is short and order.stopLoss < marketPrice then
10:                sellOrder()
```

As it can be noted, the implementation of the Stop-Loss technique is not presumed to be too complicated. However, the question arises as which value is appropriate at each moment to be set as the Stop-Loss of an order.

3.5.3 Trailing-Stop

Another simple technique, derived from the basic Stop-Loss, is the **Trailing-stop**. In general, the Trailing-Stop is based on applying the concept of Stop-Loss on the maximum benefit that an order has achieved. That is, the Trailing-Stop recalculates the Stop-Loss price of an order each time the market price exceeds the most favorable price recorded since the order was opened [37].

Given the same example as in the basic Stop-Loss case, if a trader opens a long order in the EUR/USD market at a Trailing-Stop application price of 1.1209 and sets a Stop-Loss of 2%, the price of the initial Stop-Loss will be 1.0984. In case the next price that reaches the Trader is 1.1235, the new Stop-Loss price of the order will be recalculated using this new price and the same percentage of 2% of previous risk. In this way, the new Stop-Loss price of the order will be 1.10103. If at that time the market price stops rising and falls to 1.10103, the order will be closed automatically.

The function that updates the Stop-Loss price of an order, that must be called when a

Figure 3.13: Trailing Stop

new more favorable price is detected is the same as the one described in the Algorithm 9 of the previous subsection.

Finally, the function of a Trailing-Stop algorithm for a series of prices is shown on the next page.

Although Trailing-Stop is a technique that generally has good results, we do not consider that the Trailing-Stop is the appropriate risk control system. This is due to the philosophy behind high-frequency trading that is based on buying and selling orders quickly, and not on holding them for the long-term, so this technique does not represent a great improvement over the traditional Stop-Loss. On the other hand, implementing this technique involves performing additional calculations and checks on all open orders for each new market price that arrives at the algorithm, which means a considerable increase of the response time of the algorithm.

Due to the added complexity and the little improvement that this strategy implies in comparison with the traditional Stop-Loss, we determined that this technique is not the most adequate to implement as risk control.

3.5.4 Breakeven or equilibrium point

The **Break Even** technique is a technique that, again, makes use of the definition of Stop-Loss, since it is a technique composed of this element and a new one: the breakeven point.

In this case, when opening an order, two risk control prices will be set: the Stop-Loss price, which will follow the same method as explained in subsection [3.5.2], and the breakeven or equilibrium point, which will be a price that is calculated from the entry price and the commission expenses involved in opening and closing an order, in which the order has not achieved losses or benefits.

Algorithm 11 Trailing-Stop Algorithm

```
1: function TrailingStop()
2:
3:     for each new marketPrice do :
4:
5:         for each order in openedOrders do :
6:
7:             if order.agentMode is long then
8:                 if order.stopLoss > marketPrice then
9:                     sellOrder()
10:
11:                else if order.mostFavorablePrice < marketPrice then
12:                    order.mostFavorablePrice = marketPrice
13:                    setStopLoss(order, stopLoss)
14:
15:            else if order.agentMode is short then
16:                if order.stopLoss < marketPrice then
17:                    sellOrder()
18:
19:                else if order.mostFavorablePrice > marketPrice then
20:                    order.mostFavorablePrice = marketPrice
21:                    setStopLoss(order, stopLoss)
```

$$BreakEven = EntryPrice + Commission \cdot EntryPrice \tag{3.19}$$

Using this strategy, until the market price reaches the breakeven point, the order will have as exit price the Stop-Loss. Once the market reaches or exceeds the breakeven point, the new exit price will be the breakeven [38].

Once again, we will use as an example a trader that opens a long order in the EUR/USD market when the price is 1.1209 with a Stop-Loss of 2%. When opening, the Stop-Loss price is set at 1.0984. Assuming that the commission for opening and closing an order is 0.01% for each share, the break-even point will be calculated as:

$$BreakEven = 1.1209 + 0.01 \cdot 1.1209 = 1.1321 \tag{3.20}$$

If the price reaches 1.1321, the new exit price will be 1.1321. Until then, the exit price of the order will be the Stop-Loss: 1.0984.

This strategy is too aggressive and could easily come into conflict with Coastline Trading because, due to market fluctuation, many orders would be closed without loss or profit at a given time, whereas if these orders would have been held opened, the algorithm would have achieved benefit in a short time.

For this reason, we do not consider appropriate this strategy either as the risk control system to be implemented in the algorithm.

3.5.5 Fixed Fractional Money

Another technique that many traders use to control the risk of their positions is the fixed fractional money management strategy. This strategy determines the volume of money with which an order is opened, from the risk that the trader want to assume in an operation.

The position sizing can be determined by one or another equation, depending on the trader. For instance, the order size might be determined as follows:

$$PositionSize = AvailableCash \cdot Risk\% \tag{3.21}$$

Once more, the main question is how to determine the risk that should be assumed for the orders that are opened.

By definition, this technique does not imply a real risk control of the shares, since an open order would never be closed even if the market was very unfavorable, but rather a strategy to control trading aggressiveness. In addition, its implementation in our algorithm could interfere with the position sizing adjustment that is made from the liquidity indicator (more details in the subsection 3.3.3). Thus, this technique is also not suitable to be implemented in our algorithm.

3.5.6 Risk/Reward Ratio

Finally, the last risk management strategy that is worth explaining is the **Risk/Reward ratio**. The Risk Reward Ratio is a relationship between the values of risk and the possible reward of an order that is given by a tuple of values, such as 1:2. The fact that an action has a Risk Reward of 1:2 means that the trader aims to get twice as much Reward as the Risk that assumes; see the paper of Kenton [39].

Using the example mentioned earlier, a trader opens a long order in the EUR/USD market when the price is 1.1209. If the Risk Reward Ratio of the order is 1:2, and the profit that the investor wishes to obtain is 10%, then the risk will be 5%, and the following prices will be calculated:

$$StopLoss = MarketPrice \cdot 0.95 \tag{3.22}$$

$$TakeProfit = MarketPrice \cdot 1.1 \tag{3.23}$$

The first price, the Stop-Loss, will be the one that will automatically close the action assuming losses of 5%, while the second price, the Take-Profit, will be the one that will automatically close the order obtaining the benefit of the expected 10%.

Let ExpectedProfit be the percentage value of the expected benefit of an order, which takes values from 0 to 1. The pseudocode of a function that computes the Stop-Loss and Take-Profit prices of an order from the Risk/Reward ratio, and that must be called each time a new action is opened, is the following:

Algorithm 12 Algorithm that sets the Stop-Loss and the TakeProfit prices on an order on a Risk/Reward Ratio strategy

```
1: function SetRiskReward(Order, RiskRatio, RewardRatio, Expected-
   Profit)
2:
3:     if order.agentMode is long then
4:         order.takeProfit = currentPrice * (1 + expectedProfit)
5:         order.stopLoss = currentPrice * (1 - ( RiskRatio / RewardRatio) * expectedProfit)
6:
7:     else if order.agentMode is short then
8:         order.takeProfit = currentPrice * (1 - expectedProfit)
9:         order.stopLoss = currentPrice * (1 + ( RiskRatio / RewardRatio) * expectedProfit)
```

Meanwhile the function of a Risk Reward Ratio algorithm for a series of prices would be as follows:

Algorithm 13 Risk/Reward Ratio Algorithm

```
1: function RiskRewardRatio()
2:
3:     for each new marketPrice do :
4:
5:         for each order in openedOrders do :
6:
7:             if order.agentMode is long then
8:                 if order.stopLoss > marketPrice or order.takeProfit < marketPrice then
9:                     sellOrder()
10:
11:            else if order.agentMode is short then
12:                if order.stopLoss < marketPrice or order.takeProfit > marketPrice then
13:                    sellOrder()
```

As we can see in the pseudocode above, for this technique to work, the trader must decide the value of two variables at the time of opening an order: the Risk/Reward ratio and the Expected Profit of the order. If we can determine these values by computing them automatically based on the information we have of the market, and in a way that does not interfere in the operation of the Coastline Trader, this technique is the most appropriate to implement in our algorithm.

3.5.7 Determining the Risk Management System

Once several risk management systems have been explained, the last part of this section is to determine which one of these is appropriate to be implemented in our algorithm in a way that does not interfere with its operation.

As stated in their respective definitions, some of the risk control techniques explained are not suitable to be implemented in a high-frequency trading system following a Coastline strategy. Specifically, we rule out the implementation of the risk control system using Trailing-Stop, Breakeven and Fixed Fractional Money Management techniques. Therefore, the choice of Risk Management Layer will be between a Traditional Stop-Loss strategy and a Risk Reward Ratio based strategy.

James B. Glattfelder, one of the authors of The Alpha Engine, in a discussion thread about the algorithm in GitHub [40] suggests that implementing a too intrusive Stop-Loss technique might not be a good idea, since it could interfere with the structure of the Coastline Trading.

As stated in the section 3.2, due to the Scaling Law characterized in equation 3.1, given an intrinsic event, in order to close an order, the algorithm's trading strategy will check if this order has achieved the expected profit. Therefore, every time an order is opened, the algorithm can compute the Reward price as follows:

$$Reward = MarketPrice \cdot \delta\% \tag{3.24}$$

And consequently, the TakeProfit price can be computed as:

$$TakeProfit = MarketPrice \cdot (1 + \delta\%) \tag{3.25}$$

Using δ_{up} or δ_{down} depending on the event that triggered the action.

The fact of being able to know the Reward when opening an order would justify using the Risk / Reward Ratio strategy to manage the risk. In addition, by choosing this strategy, we would not have to determine which the most suitable Stop-Loss value for each opened order is as if we had chosen a traditional Stop-Loss technique, since this value will be determined according to the threshold. So that, we will use the Risk Reward ratio to control the risk of our algorithm.

In order to apply this technique, the last question to solve is to determine which Risk/Reward ratio we want to set for each of the open orders. That is, once the algorithm opens an order and computes the Reward that it wants to obtain, what proportion, based on this Reward, should be set as the Risk of the order?

The value of the liquidity indicator $\mathcal{L}$ can be used to solve this question and determine the value of the Risk of an order. Be reminded that we introduced $\mathcal{L}$ in the section 3.3 and we explained that it is an indicator that takes a value between 0 and 1 and can be used to measure the unlikeliness of the occurrence of price trajectories. When the value of $\mathcal{L}$ approaches to 1, then the algorithm is more certain about the market behaviour. On the other hand, when the value of $\mathcal{L}$ approaches to 0, then the market behaviour is more uncertain. Therefore, it makes sense that the smaller $\mathcal{L}$ is, smaller is the risk that the algorithm should assume. In addition, it must be taken into account the fact that the Stop-Loss must not be so intrusive so it does not interfere with the Coastline trading strategy.

In order to determine the range of values the Stop-Loss should be fixed between, we have performed several simulations of the algorithm operating with historical data sets and

setting multiple Stop-Loss values. These simulations have been performed in 7 of the main Forex pairs of currencies (EUR-CHF, EUR-GBP, EUR-JPY, EUR-USD, GBP-USD, USD-CAD and USD-JPY) during the period between 01-2014 and 03-2019, giving multiple values of the original de δ and multiple values of Stop-Loss. Due to the extend of the resulting documentation, we have focused on analyzing the results that corresponds to the pairs of currencies EUR-GBP and EUR-USD, due to the fact that during this period the prices-series of these pairs showed a very different behaviour as shown in the figure below.

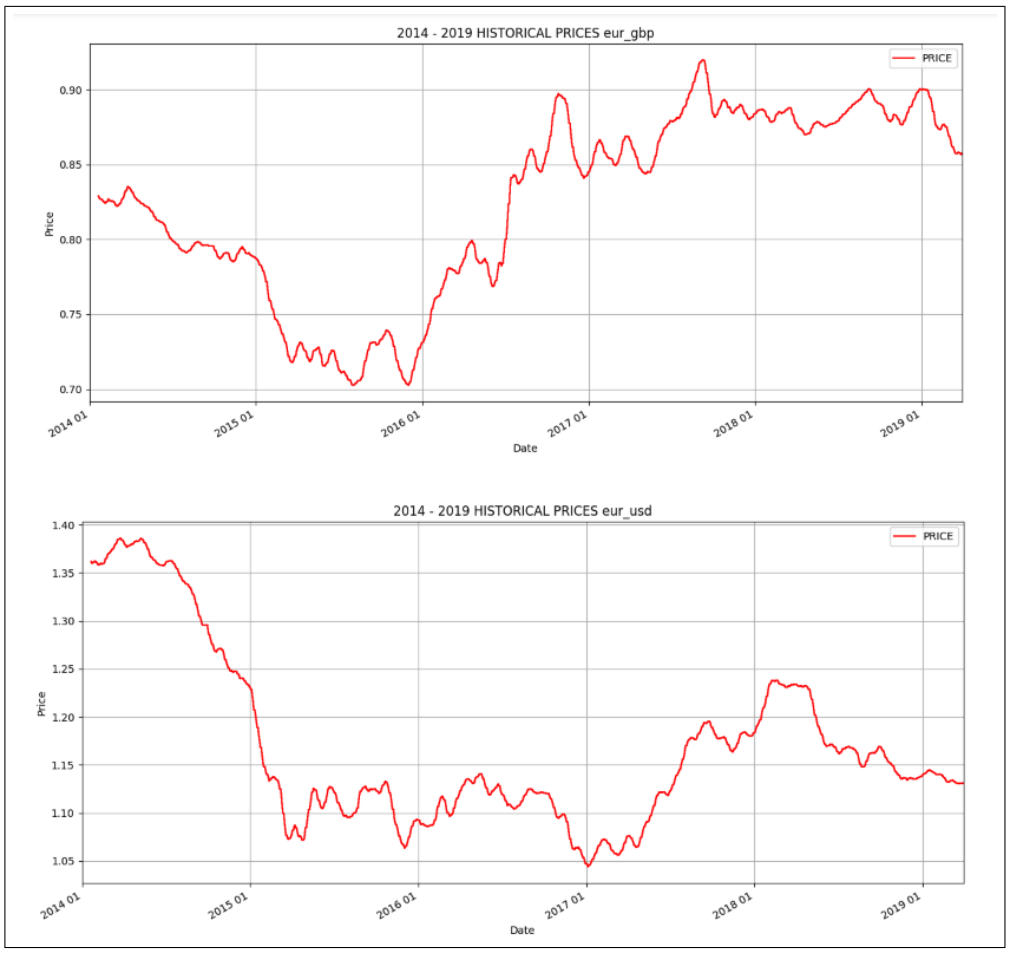


Figure 3.14: Market prices evolution between Janaury 2014 and March 2019 of the EUR-GBP and EUR-USD markets.

A summary of the executions of the runs for these currencies for the highest and lowest threshold values is shown below.

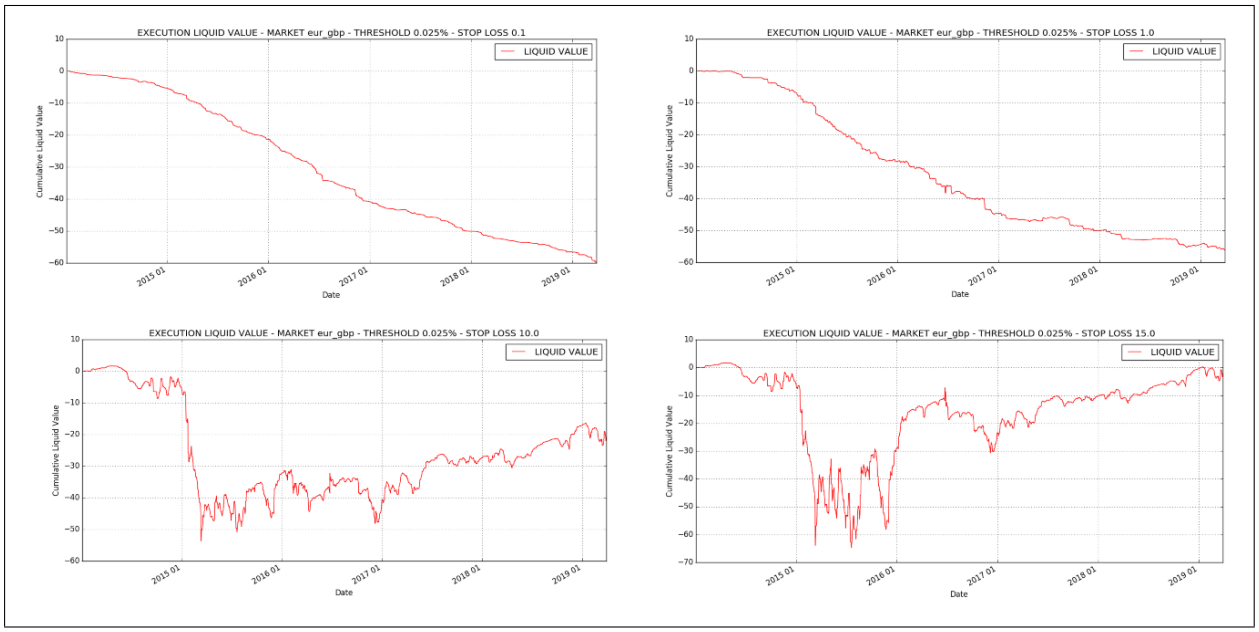


Figure 3.15: Results of the executions on the currency EUR-GBP with threshold 0.025%.

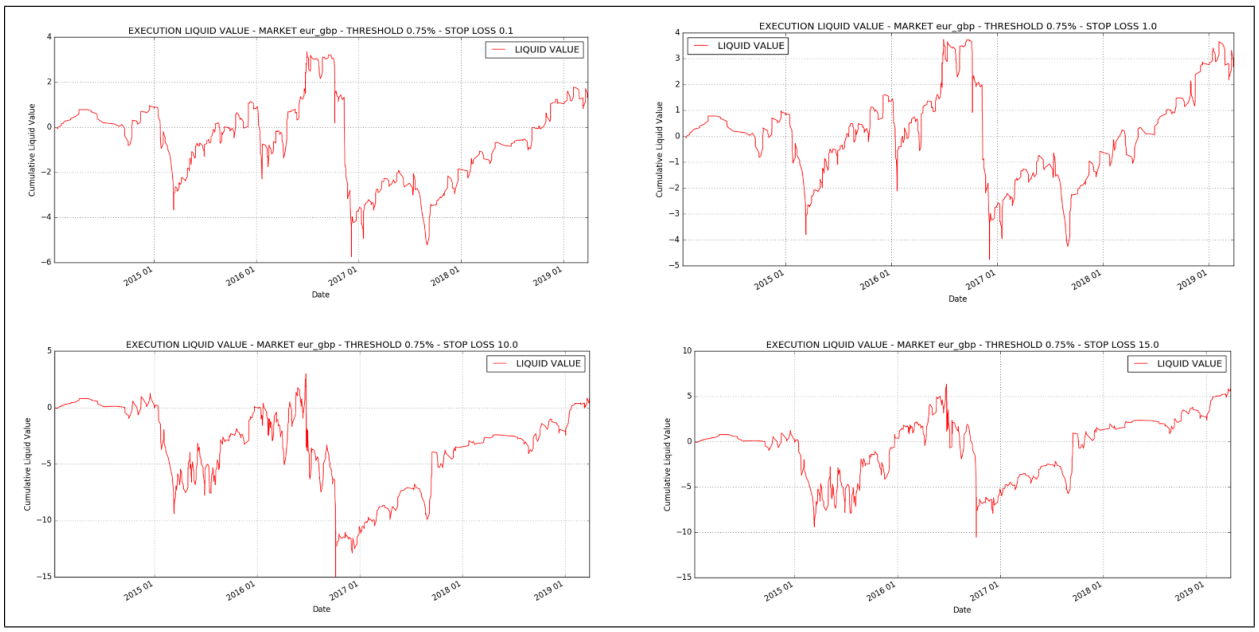


Figure 3.16: Results of the executions on the currency EUR-GBP with threshold 0.075%.

In the graphs, the y-axis represents the cumulative balance of the execution, while in the x-axis the dates associated with the balance are presented. The cumulative balance of the execution is computed as the sum of the volume and price of the opened and closed orders. The higher the value of the balance, the better the results for an execution are considered.

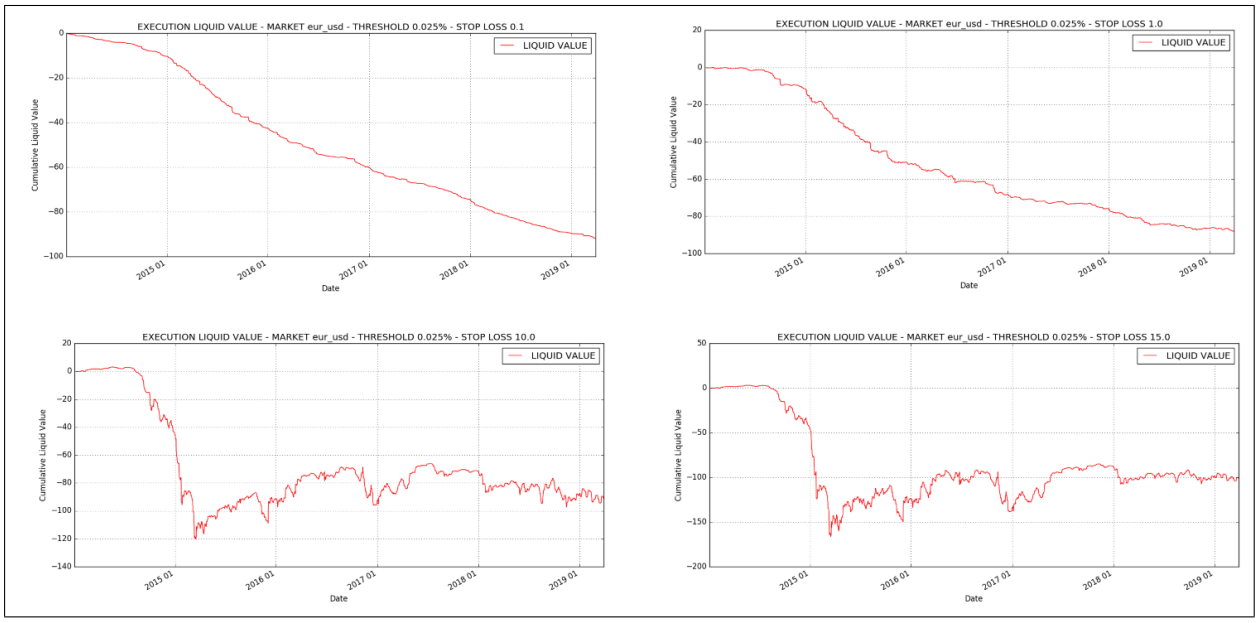


Figure 3.17: Results of the executions on the currency EUR-USD with threshold 0.025%.

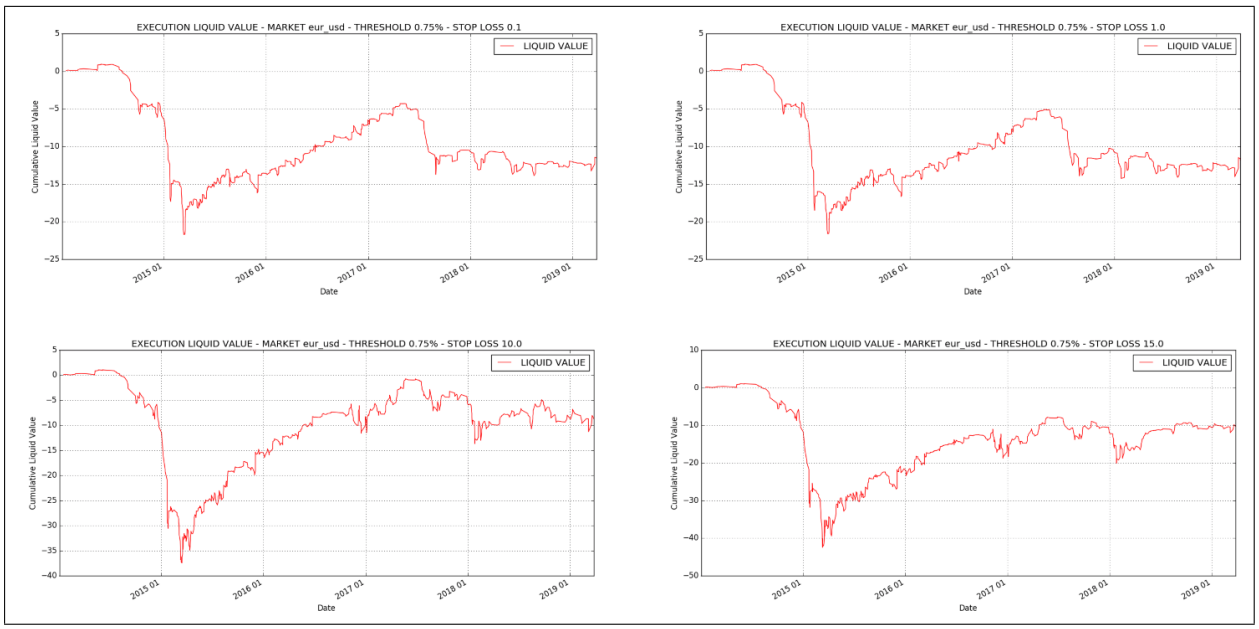


Figure 3.18: Results of the executions on the currency EUR-USD with threshold 0.075%.

From these simulations, we can note that, in the general case, the algorithm starts obtaining notable benefits from the Stop-Loss value of 10%. So that, our Stop-Loss should take a value of, at least 5%. However, we can also note that the difference between profits from a Stop-Loss of 10% to a Stop-Loss of 15% are almost null, but the lower the value, the more number of orders will be closed.

We can also note that the algorithm is very sensitive to the threshold. For instance, in the case of the EUR-GBP executions, the algorithm moves from having considerable losses to obtain profit only by changing the threshold value. This fact may be indicating that the stop loss value should also be related to the threshold value. Executions show that the smaller the threshold, worse results are obtained. Consequently, the smaller the threshold, the smaller should be the Stop-Loss value.

Taking all these factors into account, and knowing that the Stop-Loss must not be too intrusive in order not to interfere with the Coastline Trading strategy, we propose a method to automatically compute the Stop-Loss value where, ideally, it takes a value between 5% and 15%, depending on the value of the indicator $\mathcal{L}$ and the threshold that triggered the event. The lower the indicator $\mathcal{L}$ and the threshold are, the more intrusive the Stop-Loss will be and vice versa, and so, the lower the risk assumed will be. This method follows the next equation to set the value of the Stop-Loss:

$$StopLoss = ((0.15 - (0.1 - (\mathcal{L} \cdot \delta \cdot 10))) \cdot 100)\% \tag{3.26}$$

Using this method, we expect have an original threshold value between 0.01% and 1%.

So that, in the best case when the algorithm has a value of $\mathcal{L}$ of 1 and a $\delta_{original}$ of 1%:

$$StopLoss = (0.15 - (0.1 - (1 \cdot 0.01 \cdot 10)) \cdot 100)\% = 15\% \tag{3.27}$$

In the worst case, when the algorithm has a value of $\mathcal{L}$ near to 0 and a $\delta_{original}$ of 0.01%:

$$StopLoss = (0.15 - (0.1 - (0 \cdot 0.0001 \cdot 10)) \cdot 100)\% = 5\% \tag{3.28}$$

The pseudocode of the funcion used to set the Stop-Loss and Take-Profit values is shown below. This function must be called when opening an order:

Additionally, the algorithm that will manages the risk of the opened orders and will check if, given a new market price, the algorithm must sell an order because the Stop-Loss or Take-Profit has been triggered has the same shape as the Algorithm 13 (Risk/Reward Ratio Algorithm).

To summarize, throughout this section we have gone into detail about different existing techniques that are used to manage the risk of actions, in order to implement a function that allows our algorithm to determine when the loss of an order should be assumed. After the development of this feature, we conclude the development of the High-Frequency Trading algorithm.

The next and last part of this section is addressed to explain how all the features described throughout this chapter have to interact and share information between them. So that, we will try to give the reader an overview about how the algorithm carries out a real execution, since a new market price is received, until a new order is opened, or an existing one is closed.

Let the variable Event be a string that contains the event that triggered the buy of the order.

Algorithm 14 Algorithm that sets the Stop-Loss and the Take-Profit prices on an order using $\mathcal{L}$

```
1: function SetRiskManagement(Order, Event)
2:
3:     if order.agentMode is long then
4:         if event is 'downOvershoot' then
5:             order.takeProfit = currentPrice * (1 + δ_down)
6:             order.stopLoss = currentPrice * (1 - ( 0.15 - ( 0.1 - (ℒ * δ_down * 10))))
7:
8:         else if event is 'directionalChangeToUp' then
9:             order.takeProfit = currentPrice * (1 + δ_up)
10:            order.stopLoss = currentPrice * (1 - ( 0.15 - ( 0.1 - (ℒ * δ_up * 10))))
11:
12:    else if order.agentMode is short then
13:        if event is 'upOvershoot' then
14:            order.takeProfit = currentPrice * (1 - δ_up)
15:            order.stopLoss = currentPrice * (1 + ( 0.15 - ( 0.1 - (ℒ * δ_up * 10))))
16:
17:        else if event is 'directionalChangeToDown' then
18:            order.takeProfit = currentPrice * (1 - δ_down)
19:            order.stopLoss = currentPrice * (1 + ( 0.15 - ( 0.1 - (ℒ * δ_down * 10))))
```

3.6 Putting the pieces of the puzzle together

After the development carried out in the previous subsection, we can finally consider that we already have all the necessary elements to develop a completely autonomous algorithm capable of trading in the Forex market with a strategy and a model based on the mathematical investigations carried out by numerous researchers.

In a very summarized way, throughout this section we have developed the following elements:

- An event detector that, having a price series and given a threshold, using the intrinsic time concept, is able to detect changes in the market mode, update the prices taken as reference and as extreme, and determine when it occurs an Overshoot ω or a Directional Change α.
- Asymmetric Thresholds, δ_{up} and δ_{down} used to set different threshold values according to the market mode.
- A trading strategy that, in case an intrinsic event occurs, establishes if it is the right time to open or close orders.
- A liquidity indicator $\mathcal{L}$, which, based on the recent behaviour of the market, is capable of determining the amount of money with which an order should be opened at a certain time.

- A technique to calculate the market trend, using the Inventory concept as the total volume of the orders the algorithm has opened and not closed yet, and use this information to adjust the value of asymmetric thresholds and get orders to be closed easily before they accumulate.
- A novel system to manage the risk that the algorithm takes when opening an order, determining the optimal prices to close with benefits, as Take-Profit, and to close assuming losses, as Stop-Loss.

Having described these elements separately, the last phase of the development is aimed to bringing all these components together to get them to interact correctly with each other and perform the joint function of the algorithm correctly.

The details of the necessary requirements, the architecture of the software and the technologies used for the final development will be presented in the next section, so, for the time being, we will only try to give the reader an overview of how the interaction between the elements takes place. To conclude this chapter, the pseudocode below shows the final behavior of the algorithm in an execution:

Algorithm 15 High-Frequency Trading Algorithm

```
1: function TRADE()
2:
3:     for each Price in PriceSeries do
4:
5:         for each Agent do
6:
7:             intrinsicEvent = RecordEvent (Price)
8:             action = CoastlineTrading (intrinsicEvent)
9:
10:            liquidityEvent = RecordLiquidityEvent (Price)
11:            Agent.ℒ = Updateℒ (liquidityEvent)
12:
13:            if action is Buy then
14:                unitSize = GetUnitSize (Agent.ℒ)
15:                newOrder = new Order (unitSize, Agent.mode)
16:                SetRiskManagement (newOrder, intrinsicEvent)
17:                openedOrders.append (newOrder)
18:                ℐ = getInventory (openedOrders)
19:                adjustThresholds (ℐ)
20:
21:            else if action is Sell then
22:                RiskRewardRatio()
23:                ℐ = getInventory (openedOrders)
24:                adjustThresholds (ℐ)
```

Where the objects and methods called are referring to:

- Agent is an object that contains a fixed threshold, an updated value of the unique liquidity indicator $\mathcal{L}$ that corresponds to the agent and a mode that can be short or long and indicates if the orders sets to be opened are going to be long or short. There can be multiple agents trading at the same time, using different threshold values and both long and short modes.

- The function RecordEvent(Price) corresponds to Algorithm 6 and detects, using asymmetric thresholds δ_{up} and δ_{down}, if given the new market price, an intrinsic event has occurred.

- The function CoastlineTrading(intrinsicEvent) corresponds to Algorithm 2 and, if an intrinsic event has been detected in the previous function, determines the action to perform: buy, sell or pass.

- The function RecordLiquidityEvent(Price) corresponds to Algorithm 3 and detects, using asymmetric thresholds δ_{up} and δ_{down}, if a liquidity intrinsic event has occurred given the new market price.

- The function Update$\mathcal{L}$(liquidityEvent) corresponds to Algorithm 4 and updates the value of the indicator $\mathcal{L}$ in case any liquidity intrinsic event has been detected in the previous function.

- The function GetUnitSize($\mathcal{L}$) corresponds to Algorithm 5 and uses the value of the unique liquidity of the Agent to adjust the amount of money the order must be opened with.

- The object newOrder corresponds to the order to be opened and has as a parameter the amount of money invested and the agent mode that indicates if the order is short or long.

- The function SetRiskManagement(Order, intrinsicEvent) corresponds to Algorithm 14 and sets the values of the Stop-Loss and Take-Profit prices of the order just opened.

- The method append(Order) adds the order to the array of opened orders.

- The function getInventory(openedOrders) corresponds to Algorithm 7 and updates the value of the inventory $\mathcal{I}$ based on the unit size of the orders of the array of opened orders

- The function adjustThresholds($\mathcal{I}$) corresponds to Algorithm 8 and adjust the value of the asymmetric thresholds depending on the value of the Inventory.

- The function RiskRewardRatio() corresponds to Algorithm 13 and checks if any order of the array of opened orders must be sold, because it has obtained the expected profit or the assumable losses, and if this is the case, removes the order from the array of opened orders.

Chapter 4

Implementing the system

The main objective of the previous chapter was to explain and allow the reader to understand each of the elements and the basis of the development behind the operation that will implement the trading algorithm. Throughout this chapter, we will explain how and once the bulk of the algorithm is developed, we will design a real system that makes use internally of the functionality of this algorithm to operate on a real trading platform in real time. This additionally will allow us to perform tests with historical data of the Forex market.

In order to explain how the system is going to be implemented, firstly, we will define the requirements that the system must meet. Once the requirements have been defined, we will use them to design an architecture module so that the system meets the requirements. Finally, we will choose and explain the technologies considered most appropriate to carry out the process of implementation.

By the end of this chapter, we will have implemented a real system ready to be tested by a user who wants to check how the algorithm performs on historical data or on the real time market.

4.1 Requirements Specification

In this section we will identify and specify the requirements that the system implementation must meet. The requirements have been split among four categories, according to its field of action.

4.1.1 Algorithm Requirements

The Algorithm Requirements are those that define the correct functioning of the trading logic. The correct implementation of the developments described in Chapter 3 should be enough to meet all these requirements.

The Algorithm Requirements defined are listed below:

- The algorithm must be able to obtain **bid and ask prices** for every new market price.

- The algorithm must be able to manage an **indefinite number of agents**.
- The algorithm must store the **value of the threshold**, **mode and liquidity** of all of its agents.
- Given a price and a threshold, the algorithm must be able to detect if an **intrinsic event** has occurred.
- In case of an intrinsic event, the algorithm should be able to determine whether any **orders** should be opened or closed.
- The algorithm must be able to detect intrinsic events of the **liquidity** given a price and a threshold.
- In case of an intrinsic liquidity event, the algorithm should update the **agent's liquidity**.
- When opening an order, the algorithm must be able to determine the amount of money with which it will open depending on the value of the liquidity $\mathcal{L}$.
- The algorithm must be able to determine the value of **Stop-Loss** of an order based on the market price and the liquidity $\mathcal{L}$.
- The algorithm must be able to determine the value of **Take-Profit** when opening of an order based on the market price and the threshold.
- The algorithm must be capable of updating the value of the **inventory** after opening or closing an order.
- The algorithm must be able to update the values of the **asymmetric thresholds** when the inventory is updated.

4.1.2 Functional Requirements

The Functional Requirements are those that define the actions that the system must be able to perform by interacting with other systems in the real world.

The Functional Requirements defined are listed below:

- The system must be able to connect with a real broker.
- The system must be able to obtain prices in real time from a broker.
- The system must be able to perform algorithm executions with prices in real time.
- The system must be able to open orders in a real broker.
- The system must be able to close orders in a real broker.
- The system must be able to connect to a database with historical prices.

- The system must be able to perform algorithm executions with historical prices.
- The system must be able to perform algorithm executions using historical prices of the Forex market.
- The system must have an interface that allows the user to customize the executions.

4.1.3 User Requirements

The User Requirements are those that define the actions that an end user of the system can perform. I.e., the User Requirements define the possible scenarios in which the user may take part, as well as the customization options of the parameters used by the system.

The User Requirements defined are listed below:

- The user must be able to choose between real-time or historical execution.
- The user must be able to enter the credentials of his personal broker account.
- The user must be able to choose the currency with which he wants to operate in real time.
- The user must be able to choose the currency with which he wants to perform historical executions.
- The user must be able to choose the threshold with which he wants to perform historical executions.
- The user should be able to consult the results of the balance once the historical executions have finished.
- The user must be able to consult the statistics of opened and closed after the historical executions.

4.1.4 Data Model Requirements

The Data Model Requirements are those that define how to carry out the design and implementation of the Database and the Dataflow between the Database and the logical module of the system.

The Data Model Requirements defined are listed below:

- The database should store historical prices of the currencies with which they want perform to historical execution.
- The database must store the orders that are opened along with the current data From the market.

- The database must store the orders that are closing along with the current data From the market.
- The database must be able to calculate, based on open or closed orders, the total balance after an execution.
- The database must be able to calculate, based on open or closed orders, the statistics after an execution.

4.2 Software Architecture Design

Once the requirements have been defined, the architecture of the software system should focus on meeting them. Specifically, the Algorithm Requirements should have been met with the developments carried out throughout Chapter 3, so this section will be aimed to satisfying the Functional Requirements, while in the following sections we will focus on the User Requirements and the Database Requirements.

The software architecture must define the modules of the software system and how they are going to be connected for the system to perform its functionality properly. From the definition of the architecture, the final functional software will be designed.

To document the architecture of the software to be developed, we will use the 4+1 Architecture Views Model. The 4+1 Views Model is a model created by Phillipe Kruchten [41] used to document the design and the implementation of the high-level structure of a software system, prior to the beginning of the development. This model defines five concurrent views, each one referring to a set of interests of different stakeholders of the system.

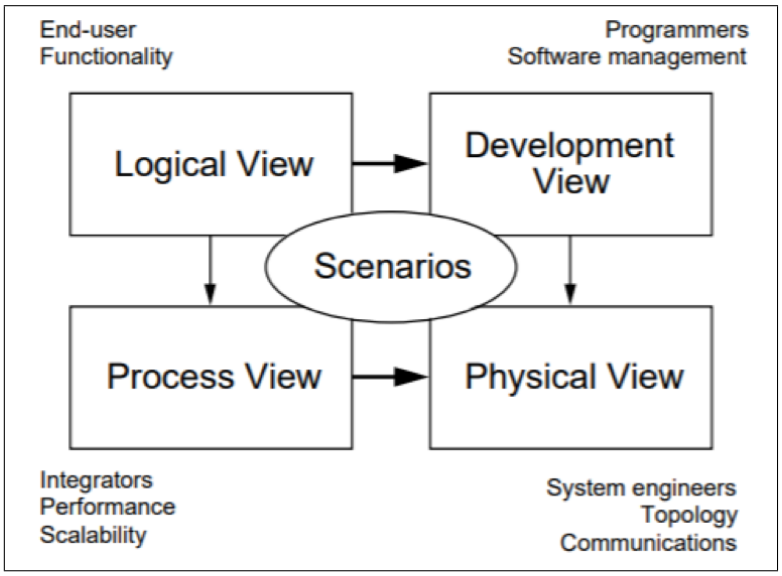


Figure 4.1: The 4+1 View Model [41].

4.2.1 The Logical View

The Logical View of the system represents the main concepts needed to support the functionality required by the system. To design this view, we will use an object-oriented method and use the Unified Modeling Language (UML) [42].

The Logical View defined is shown bellow.

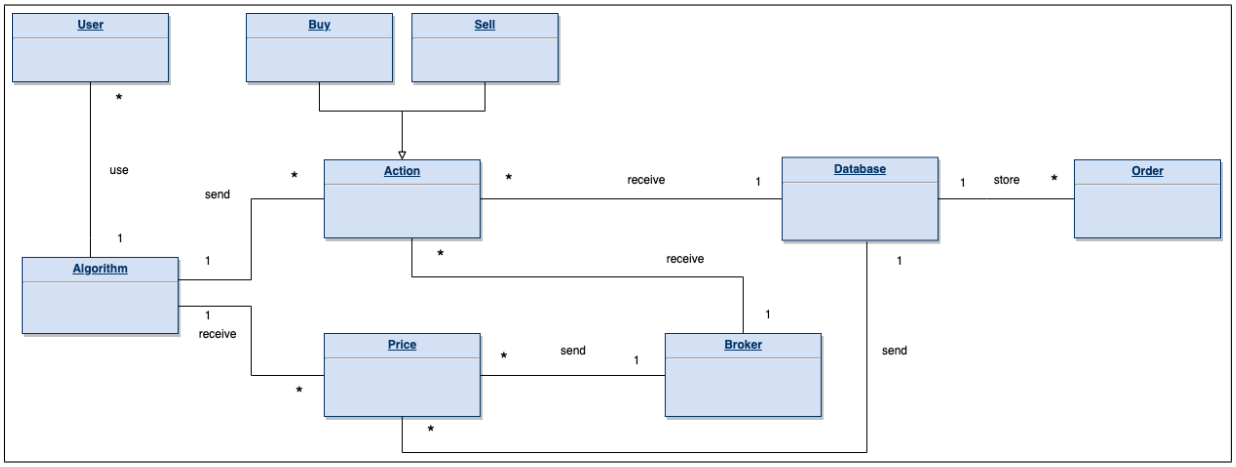


Figure 4.2: The Logical View.

From this view, we define the objects needed to program the algorithm which will implement all the logical modules of the program.

4.2.2 The Process View

The Process View describes the aspects of concurrency and synchronization of the system components. This view shows how communication between the objects defined in the previous Logical View is carried in order to provide a service.

The Process View defined is shown bellow.

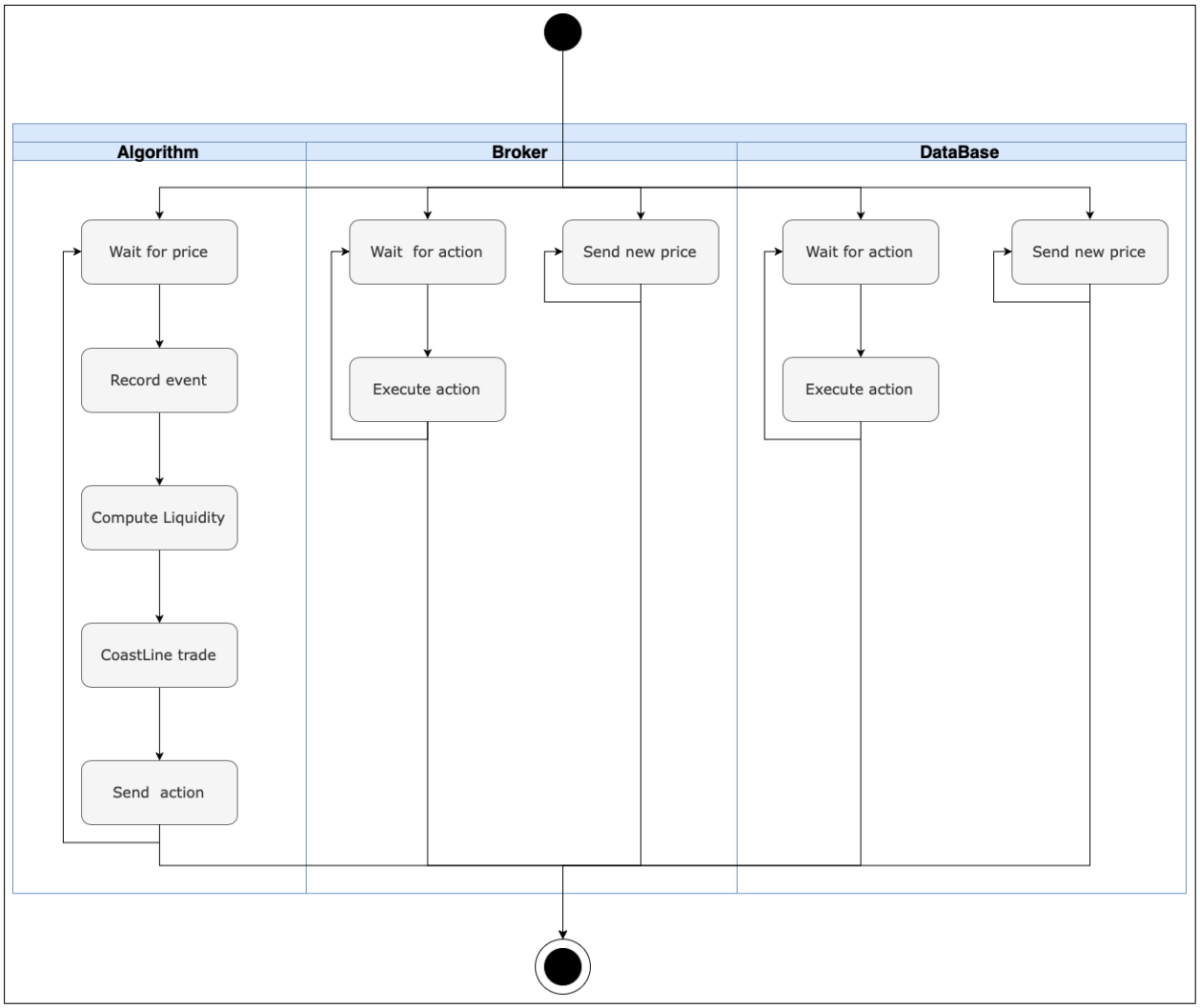


Figure 4.3: The Process View.

4.2.3 The Development View

The Development View is directly influenced by the architectural pattern chosen to be applied for the construction of the system.

The Development View defined is shown bellow.

To design this view, we have chosen to use an architectural pattern based on layers combined with the computing model Client/Server. So that, the architecture has 3 main layers:

- **Presentation Layer**: Layer that includes the components in which the interaction between user and system takes place.
- **Logical Layer**: Layer that includes all the controllers responsible for carrying out the

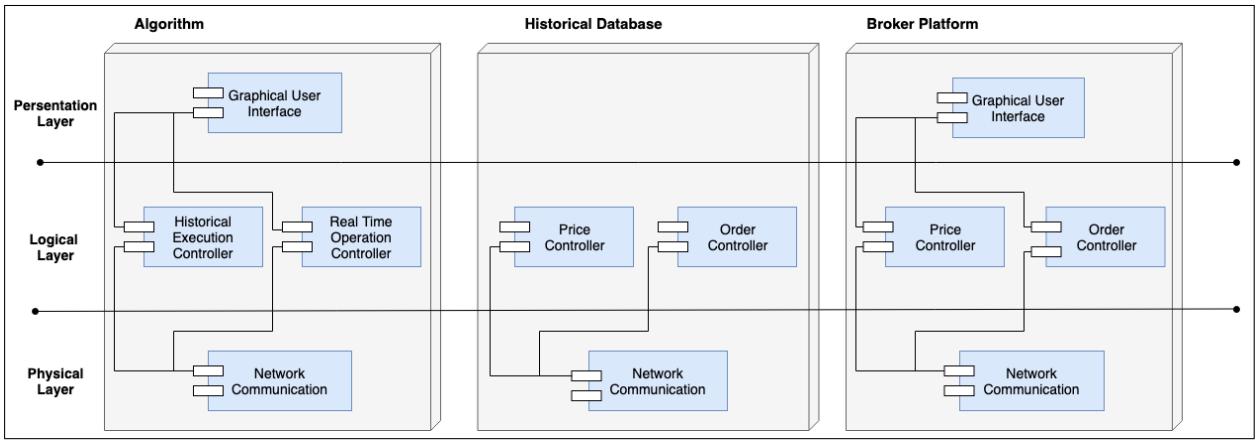


Figure 4.4: The Development View.

price and order analysis and other main actions of the algorithm. As its name suggests, this layer contains all the logic of the program.

- **Physical Layer**: Layer that includes all the components responsible for establishing connection between the modules.

4.2.4 The Physical View

The Physical View describes the mapping of the software on the hardware and reflects the distribution aspects. It represents the physical elements that make up the system and the way in which these elements are related.

The Physical View defined is shown below.

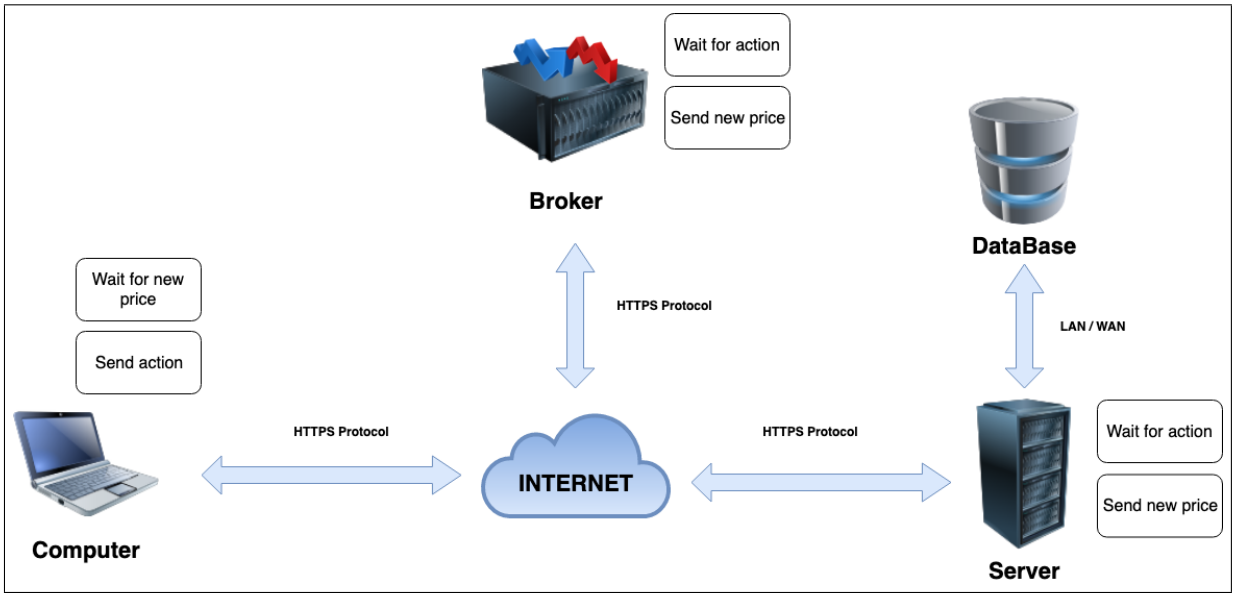


Figure 4.5: The Physical View.

4.2.5 The Scenarios View

Finally, the Scenarios View shows the main actions the user can perform on the system. For the design of this view, we have chosen to define the main system use cases.

The Scenarios View defined is shown below.

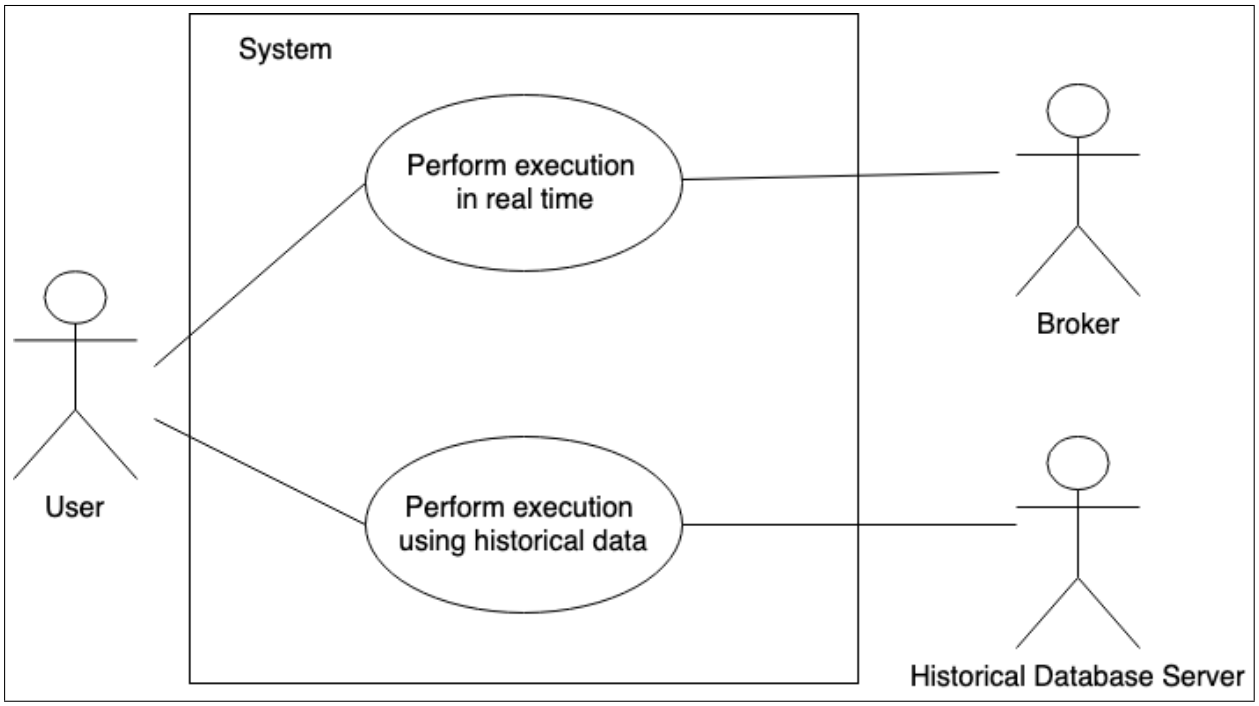


Figure 4.6: The Scenarios View.

4.3 Development of the user interface

In order to make the system easy to use for anyone, we have developed a basic user interface for both historic trading and live trading.

If the user chooses the live option, he will be requested to introduce a broker api token and a pair of currencies in which to trade. After that, the algorithm will automatically start trading in real time. We recommend the user to monitor the orders that are being bought and sold by the algorithm, as well as his balance and other information, via the broker website, which has a good GUI.

On the other hand, if the user chooses to trade with historic data, he will be requested to introduce a pair of currencies, a start, an end year, and a threshold value. After this the algorithm will connect with our historical data base and start trading. It is important to mention that, as the threshold gets smaller and the selected years get bigger, i.e. (threshold=0.0015, start year=2001, end year=2019) the duration of this process can last up to 90 minutes. Once the process is completed, the user will be able to see the stats generated, and a graphic of the liquid value evolution during those years.

It is worth mentioning that this is a basic GUI, and it allows the user to introduce invalid data or perform invalid actions. Therefore, it is recommended to make a "favorable" use of the GUI. When calculating results from the historic data base, if the mouse pointer is loading, the program has not crashed, it is running accordingly and it will display the results

soon, be patient.

We can see the screens sequence of a historical execution and a live execution on the next few pages:

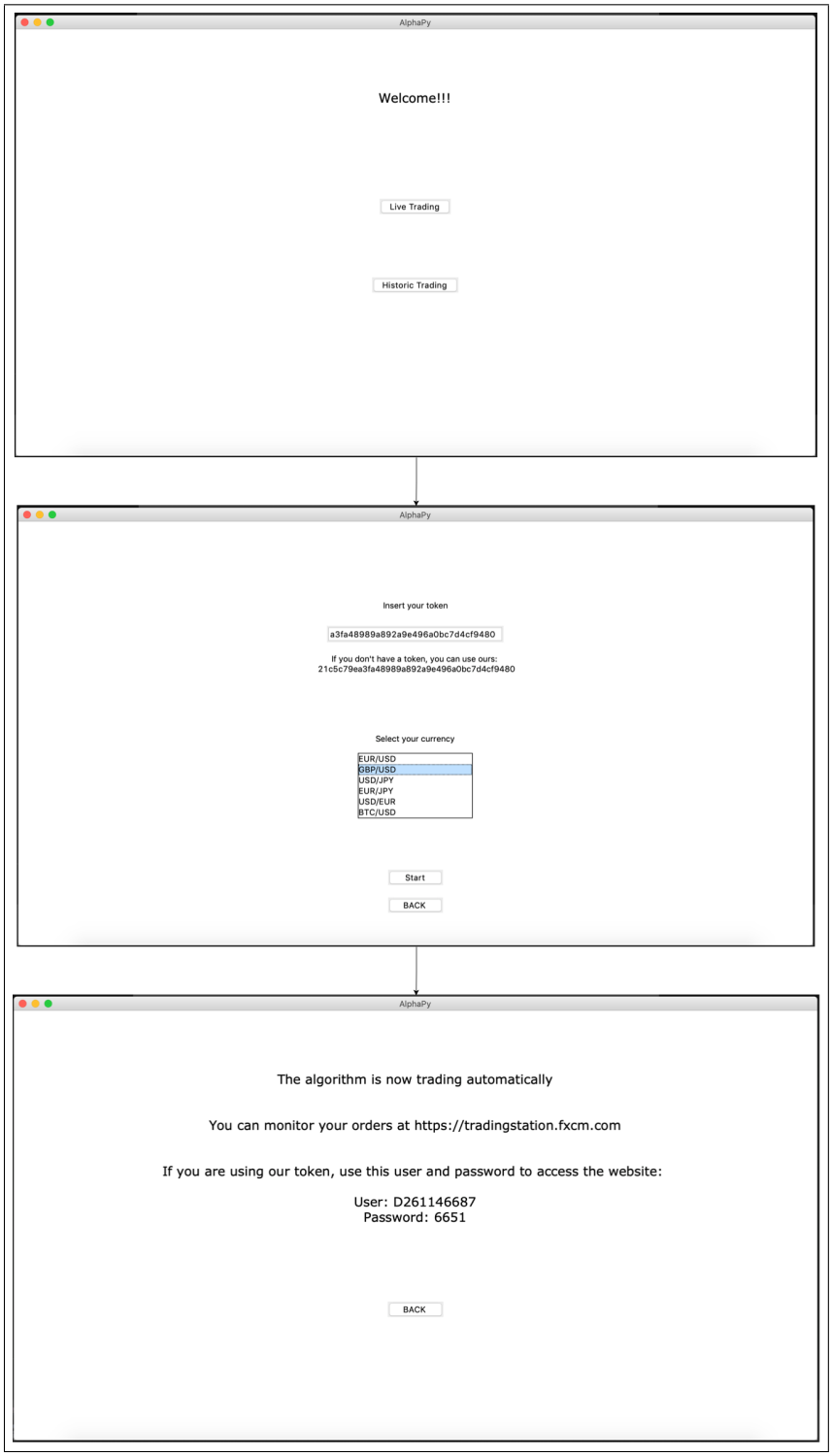


Figure 4.7: Screenshots from the user interface (I).

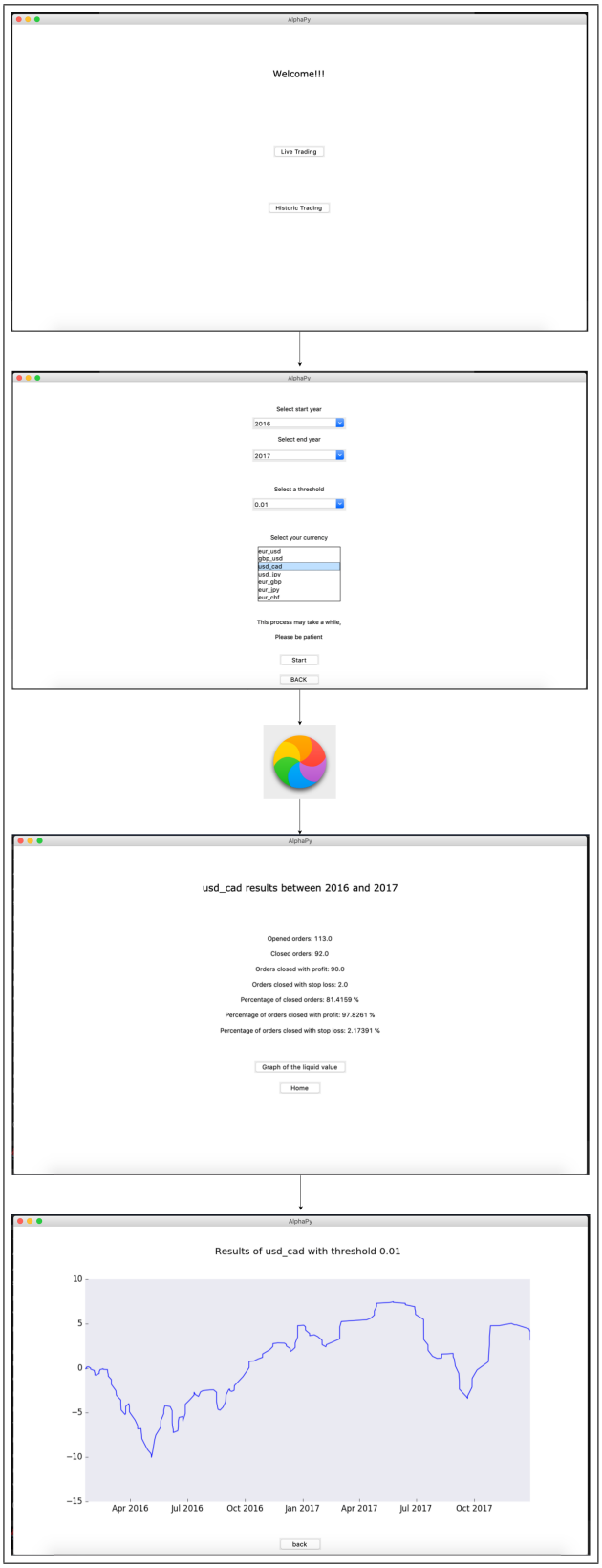


Figure 4.8: Screenshots from the user interface (II).

4.4 Defining the Data Model

In this section, we will design a Data Model with which to meet the Database requirements defined at the beginning of the chapter. When the algorithm establishes connection with a broker platform to perform an execution in real time, we expect to obtain the market price data from the broker's own data systems, and expect this system to stores the orders that the algorithm places through a connection to the broker's API. However, to perform executions using historical data, the algorithm must be provided by data by our system. Therefore, our data model is focused on supporting executions with historical data.

To design the data model, we have to take into account that the Database, in addition to store historical prices and the opened and closed orders, must use the data to compute the balance results and execution statistics after an execution.

The design of the Data Model is shown below:

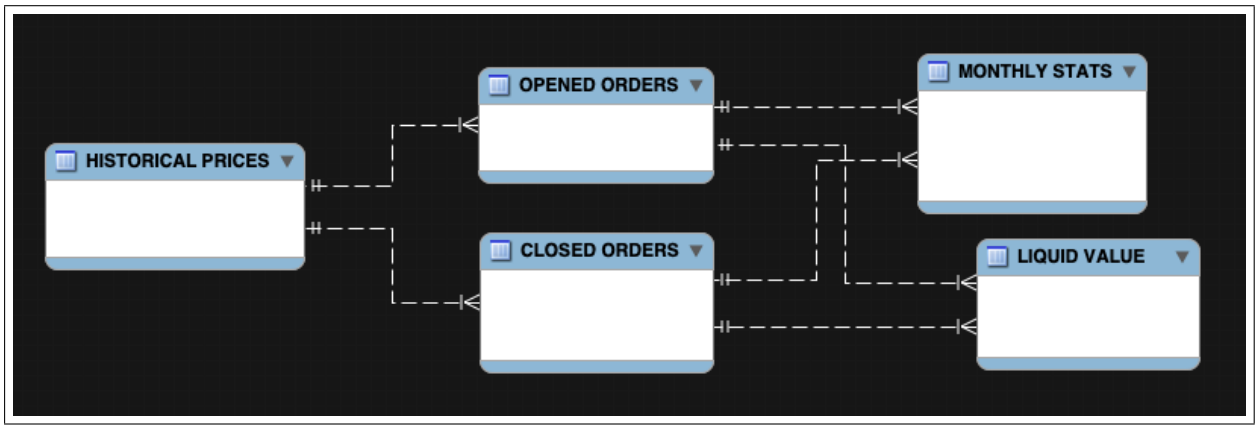


Figure 4.9: Designs of the tables and the relationships between them that the Datamodel must contain.

The function of each of the tables shown in the design of the data model is briefly described below:

- Historical Prices: this table must contain information about bid and ask historical prices of the market, and the date and time the price was registered. With the aim of having appropriate data for the operation of a high-frequency algorithm, the data frequency of the prices to be stored must be one price per minute and the period will be between January 2001 and March 2019.

- Opened Orders: this table must contain all possible information about the orders set to be opened: open price, date and time, volume, liquidity, Stop-Loss price, Take-Profit price, long or short mode, event that triggered the buy, etc. This table will refer to Historical Prices table using the open price. Once the order is closed, it must be removed from this table.

- Closed Orders: as in the Opened Orders case, this table must contain all the information of the order when this order is set to be closed, adding some information such as if the order was closed by Stop-Loss or Take-Profit. This table will also refer to Historical Prices table using the close price. Additionally, this table will refer the Opened Orders table storing its Original Id.

- Liquid value: this table will be update when the algorithm opens or closes an order, and will update the current liquid value of the execution up to that time.

- Monthly Stats: this table will also be updated once the algorithm finish analyzing all the historical prices of a month, and will store in tables some execution statistics as the number of opened and closed orders, the percentage or closed orders, the percentage of Stop-Loss or the percentage Take-Profit orders that has taken place in a certain month. After the whole execution, the table will be updated, adding a row with the total statistics results.

These tables and relationships must be created for each pair of currencies that the system must work with. In other words, the Data Model will have unique tables of historical prices, opened orders, closed orders, order movements and execution statistics for each pair of currencies with which we want to perform simulations of the algorithm.

The data flow between the algorithm and the database follows the diagram below when the algorithm performs an action (buy or sell) is shown below.

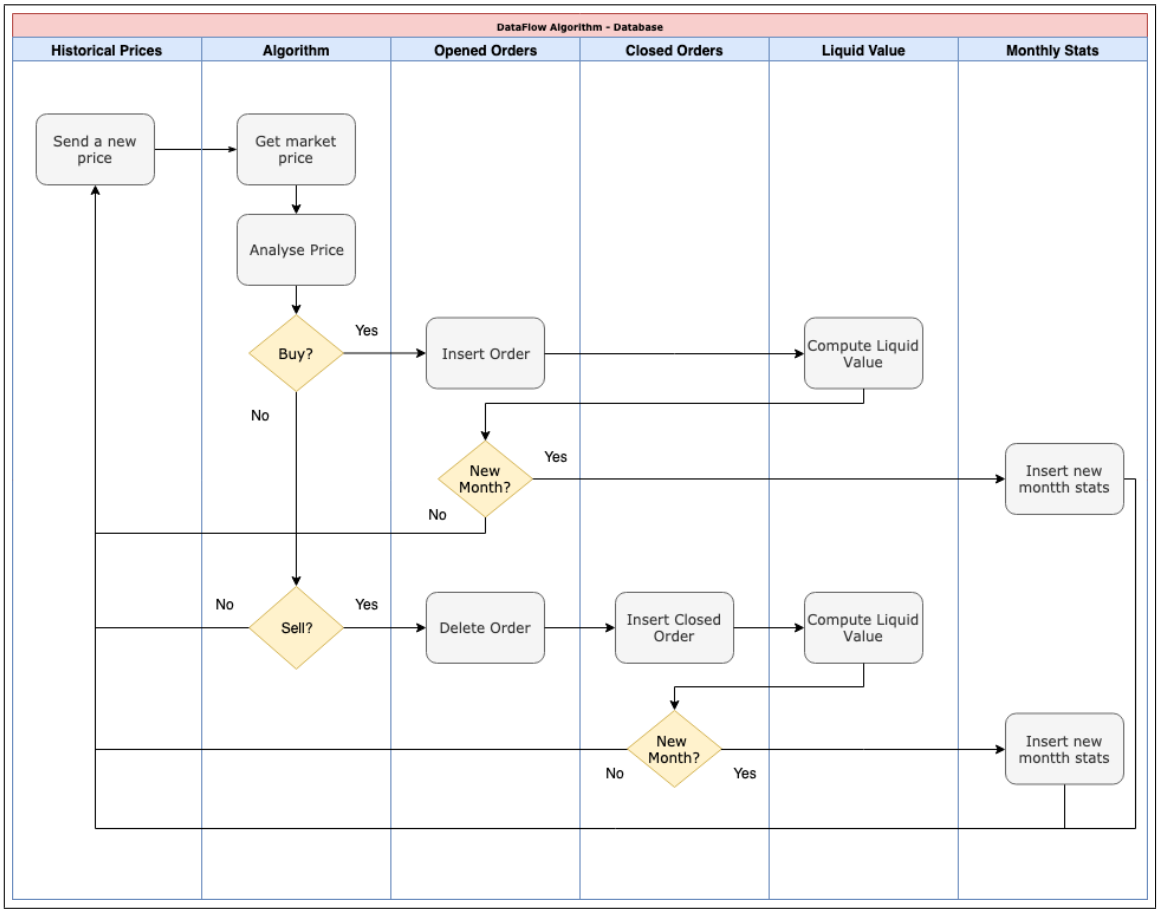


Figure 4.10: DataFlow Algorithm - Database.

With the aim of creating a Database that is not too big and difficult to migrate, we have chosen only seven of the main Forex currencies to store data. These seven currencies are EUR-CHF, EUR-GBP, EUR-JPY, EUR-USD, GBP-USD, USD-CAD and USD-JPY.

It must be noted that, if a user wants to perform simulation on another pair of currencies, he can provides his own data feed.

4.5 The Technologies of the Implementation

In the previous sections, we explained the models that will be used for the implementation of the architecture, the interface and the data model necessary to start up the system and meet all the requirements. In this section, we will explain the technologies that are considered most suitable for those implementations to be carried out. Additionally, we will try to give the reader a general idea about how the diagrams and models defined in the previous sections will be implemented using the chosen technologies.

Looking back, it was mentioned earlier that, if the development guidelines explained throughout Chapter 3 are followed, the algorithm resulting from this development should

meet all the Algorithmic Requirements defined.

It must be taken into account that the nature of a high-frequency trading algorithm requires a great speed of analysis of prices and execution of actions in order to achieve an optimal response time. In addition, the algorithm will have to process a large amount of data, perform complex calculations such as the liquidity indicator, connect to a database and graph historical results that can count on hundreds of thousands of data.

Based on these questions, the programming language chosen to carry out the implementation is Python 3. Although some other languages as Java can offer a more optimal response time in certain operations, we consider Python the most appropriate language due to the high number of libraries that ease the development of such complex system. The libraries used for the development of the operation of the program are:

- The Math library: provides access to the mathematical functions defined by the C standard. This library is mainly used to compute values such as the Liquidity Indicator $\mathcal{L}$
- The Pandas library: provides high-performance easy-to-use data structures and data analysis tools. This library is used to process the results and statistics of the executions.
- The Matplotlib library: is a plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments. This library is used to plot the results of the executions.
- The PyMySQL library: provides the needed driver to connect Python code to a database. This library is used to perform the connection to the historical prices database.
- The Tk library: provides a robust and platform independent windowing toolkit that can be used to design graphical interfaces that interacts with the python program. This library is used to develop the user interface which the user can perform actions in the system with.
- The pyTest and Coverage libraries: provide a framework that makes it easy to write small code tests. These libraries are used to perform the unitary tests of the logic of the algorithm.
- The FXCMpy library: provides a RESTful API to interact with the FXCM trading platform [43]. Among others, it allows the retrieval of real time streaming market prices data. This library is used to establish the connection between the algorithm and the real time trading broker platform.

The final class diagram of the objects used by the implemented algorithm is shown below:

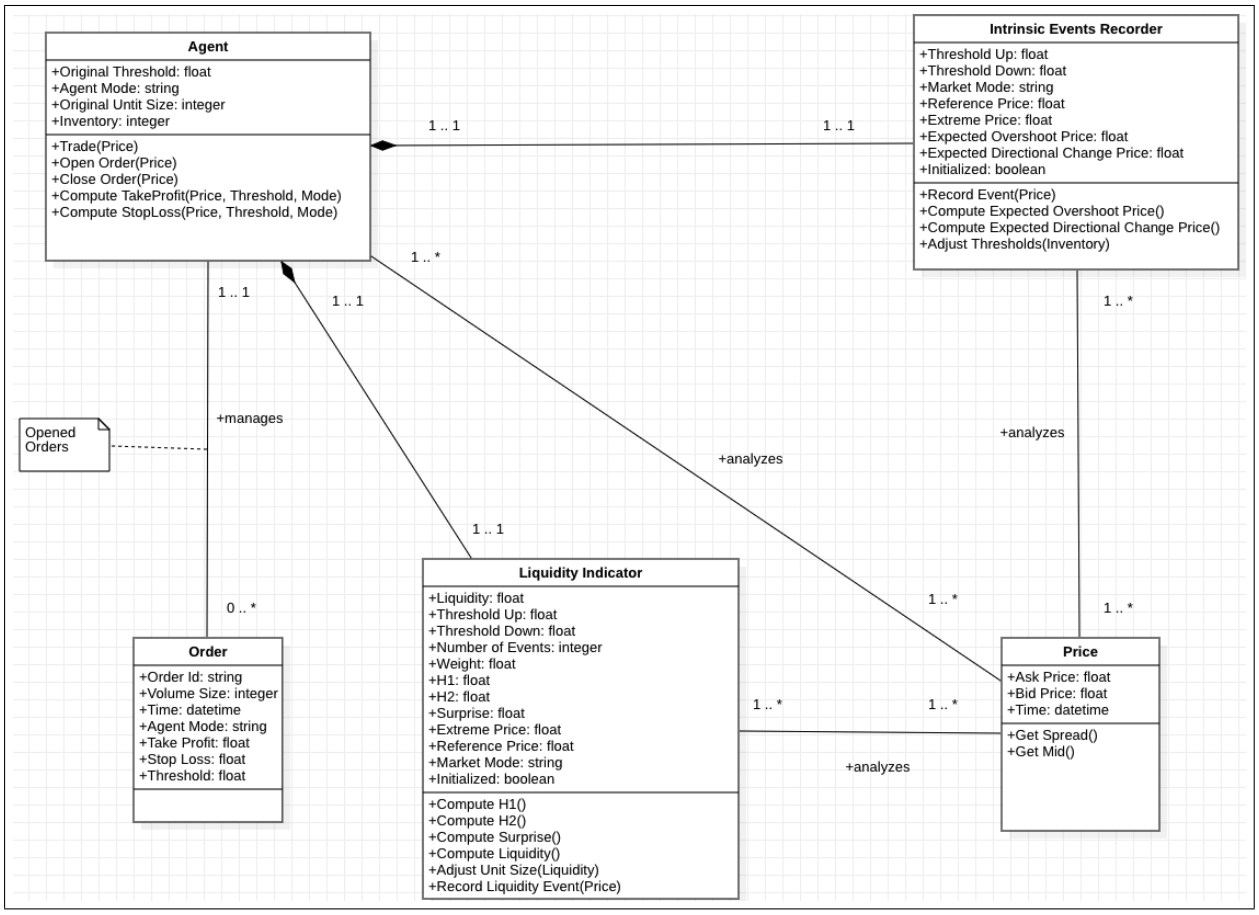


Figure 4.11: Class Diagram of the objects, attributes and methods of the Implemented Algorithm.

In Section 4.4 we defined and designed the Data Model. In order to implement this Data Model meeting the Database Requirements and adequately performing the Data Flow, it has been created a relational database using SQL technology. We have opted for a relational database instead of a non-relational database for several reasons:

- It provides a better performance than non-relative databases when inserting, updating and deleting data. Throughout an execution of the algorithm, tens of thousands of orders are selected updated and deleted from the database, so performance will be a key point to choose for SQL technology.

- We have extensive prior knowledge in SQL technology that allows us to develop the required actions in an easier way.

- Currently, most commercial servers offer the migration of SQL databases, but just very few offer to store non-relational databases. In addition, the servers that offer this service represent a significant increase in the monthly price.

- The connection from the algorithm to the database can be carried out in a very simple way from the code when the database is SQL, since the existing libraries have extensive documentation.
- Ease of configuration so that the data can be consulted, inserted and modified by the users of the program without the need to assign special permissions.
- Possibility of guaranteeing the data validity even in the event of errors, power failures in the database due to ACID properties (Atomicity, Consistency, Isolation, and Durability) [44].

To sum up, a relational database complies with all the necessary requirements, is inexpensive and provides a good performance in the data processing during the executions.

As previously stated, to correctly integrate the database in the algorithm operation, it has been used the PyMySQL library. This library allows to connect the database from the system execution environment and execute SQL sentences from the code. We can select the data, perform insertions of orders and statistics, update the rows stored and delete the opened orders when they are set to be closed using algorithm objects. For instance, we can insert into the 'Opened Orders' Database table an order declared in the algorithm, which contains internal data such as the current liquidity indicator or inventory values.

In order to the Historical Prices Database to be consulted by any user through an internet connection, the database will be uploaded to an online server. The connection is automatically made from the Python code when historical tests are executed.

The implemented data model for one currency pair, after determining which fields should be stored in each of the tables is shown below. Recall that it must be replicated for all the pairs of currencies we want to store.

Finally, to support the main functionality of the algorithm, apart from running tests with historical data, it is necessary that the program establishes a connection with a trading platform that allows it to obtain market prices in real time and opening and closing orders simulating the behaviour of a real trader. As established in the architecture of the system, this connection must be made through the HTTPS protocol. In other words, an external service to our system must provide us with market data through an API connection that uses the Internet. The trading platform chosen to make this connection is FXCM.

FXCM, also known as Forex Capital Markets, is a retail trading broker in the foreign exchange market, based in London. Accessing to the FXCM database which stores data from multiple currency prices is public and free. In addition, this broker provides all the necessary services to be used by our algorithm. The fact that the broker manages orders internally in its server and that the algorithm can consult them by only making calls to the API, means that our program does not need to store the opened and closed orders or the statistics and balance data, what supposes a big reduction in the operational complexity. That is, the algorithm will only have to handle of receiving the last market price, analyze it and inform the broker if an order should be opened or closed.

In addition of providing these services, FXCM also provides a user interface on its website. This interface allows the user to consult the orders that have been opened, closed and other

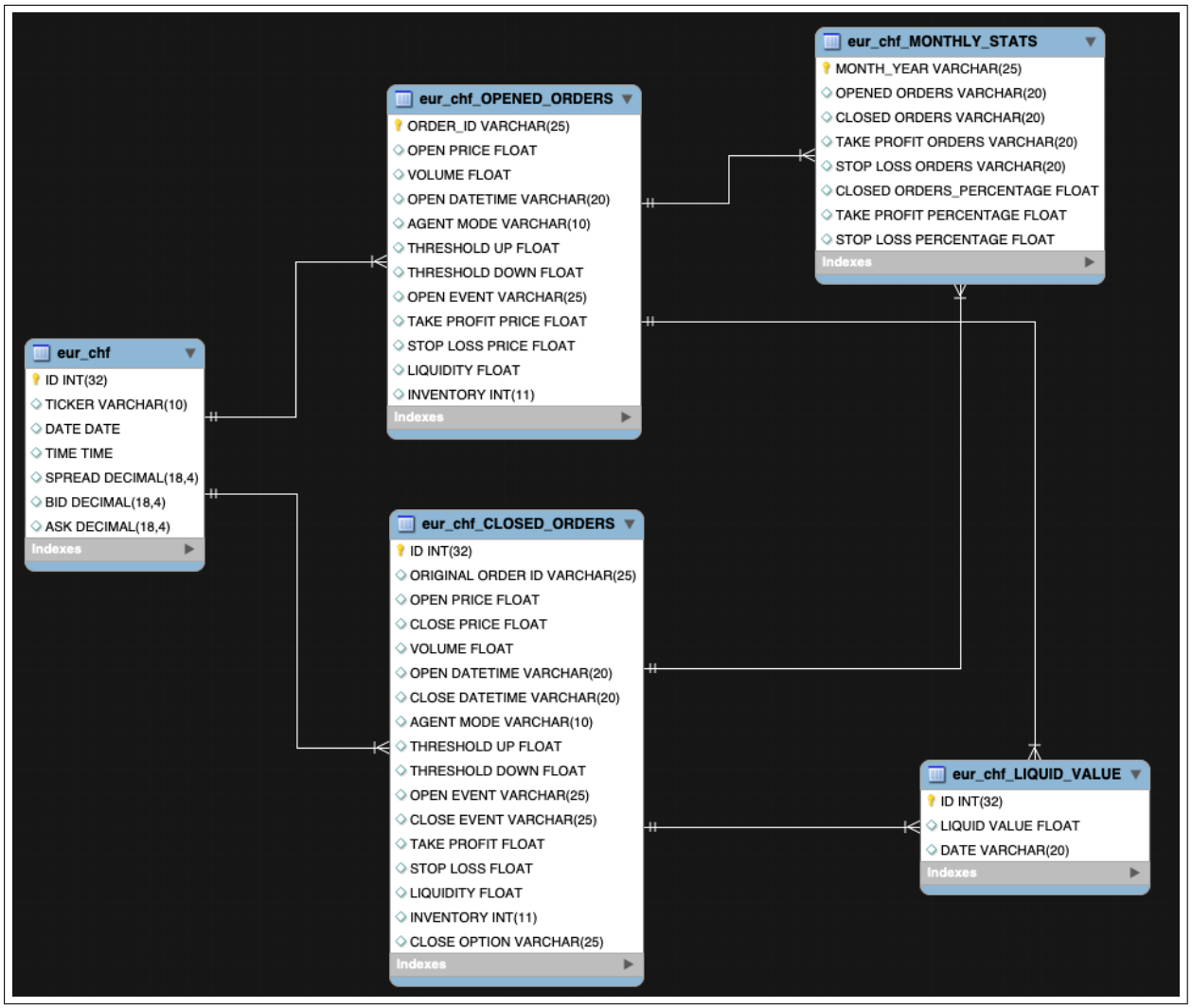


Figure 4.12: Implemented DataModel tables and fields stored.

interesting data such as the balance of the account or the recent evolution of the markets.

Therefore, a user who wants to use the algorithm as a real trading system, will only have to open an FXCM account and enter his Token API. As the action of creating a Demo account is allowed without the need to enter real money, the user will be able to test the algorithm in real time without incurring any kind of expense. However, if a user also wishes to operate with real money he can.

Operating in real time, it has been observed throughout various executions that the algorithm is very sensitive to the chosen threshold value. In order to the user not to choose an atypical or not optimal threshold value, the algorithm will always use the same number of agents and the same favorable thresholds. Therefore, when the algorithm is executed in real time, the following agents will be trading:

- A long and a short agent with a threshold of 0.01%

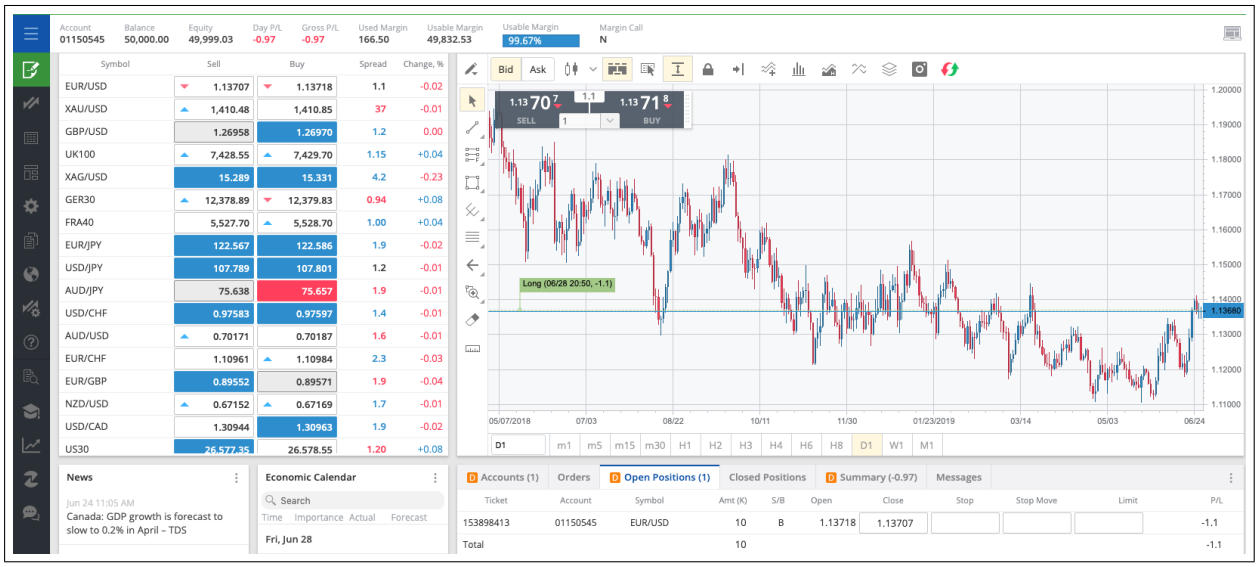


Figure 4.13: Graphical User Interface of the FXCM Broker.

- A long and a short agent with a threshold of 0.25%

To sum up, so far, we have implemented a Python program that is able to run a high-frequency trading algorithm either using historical market prices, by connecting to a SQL database, or trading in real time market with a Demo or real account, by connecting to FXCM broker platform.

Having done that, our system meets all the defined requirements and is ready to be used in the real world. However, before putting an end to the project too early, in the next chapter we will try to ensure, by performing multiple tests, that both the development and the implementation have been carried out correctly and that the final product is ready to be used.

Chapter 5

Testing and Verification

After the Development and Implementation of the System phases, the last part with which we will conclude the Project is the Verification of the system behaviour. The Verification phase is carried out by conducting tests. These tests are performed, not only to validate the correct operation of the system, but also to check the efficiency of the algorithm compared to The Alpha Engine. For that purpose, we have split this section into five parts, each of them clearly differentiated from the others and with its own objective:

- First, it will be performed unit testing for the components of the main algorithm. I.e., we will program unitary tests using tools of automation to ensure the elements of the algorithm are working correctly and as expected.
- Secondly, tests will be carried out to verify that the data is being stored correctly in the Database. In this part we will ensure that the data is being communicated correctly between the algorithm and the Database.
- Thirdly, user tests will be carried out to verify that the user interface is working as it should and that the data the user gives is being communicated correctly to the algorithm.
- Fourth, efficiency tests will be carried out among the different versions of the algorithm. These tests are made in order to compare which of the development versions strategies obtain a higher return.
- Finally, we will perform real executions to verify that the algorithm works correctly with the trading platform. That is to say, the algorithm obtains real time price data and uses it to detect intrinsic events and buy or sell shares automatically.

5.1 Unitary Testing

In order to ensure the proper functioning of the algorithm, the first thing we must to verify is that each of the elements work well individually. To test individually the components of a software product, unit tests have traditionally been carried out.

A unit test is a method to check the correct operation of a code unit. For example, in object-oriented programming, the unit test must check the operation of the methods of a class separately. The unitary testing serves to ensure that each unit functions properly and efficiently. In addition to verifying that the code does what it is supposed to do, with unit tests it is also verified that the method name, the attributes, variables and types of the parameters and the returns are right.

Due to this reason, this first testing section shows how the automated unit tests are going to be performed for each of the algorithm elements developed in Chapter 3.

Therefore, the components that must be tested are the logical classes and methods of the algorithm. These classes and methods were described in the previous chapter in the class diagram (Figure 4.11).

The unit test will be coded in Python language using the library Unittest and the tool Coverage.py to graphically obtain a report of the lines of code that have been covered by the unit tests.

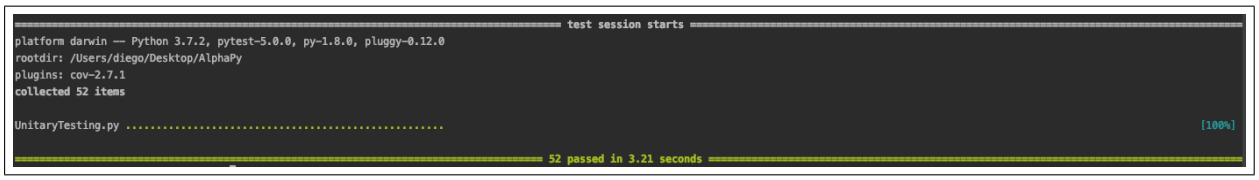


Figure 5.1: Execution of the Unitary Testing file.

The results of the unitary testing phase are shown below.

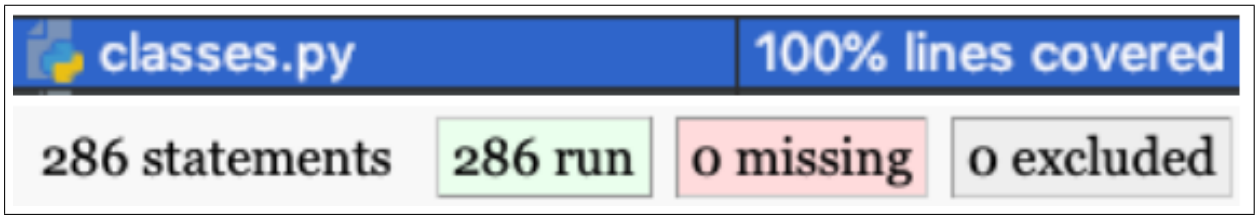


Figure 5.2: Results of the generated Unitary Testing coverage report.

As the reader can see, all the elements and methods of the file classes.py, which contains the objects Agent, EventsRecord, LiquidityIndicator, Price and Order, have been tested successfully. So, the 100% of the code lines are covered and correctly tested.

5.2 Data Model Testing

The Datamodel testing helps us to ensure that the connection between algorithm and Database is being done correctly. I.e., we will verify if the prices are being obtained correctly from the database, if the orders are being stored correctly, if the tables are updated when an

order is closed, and If the statistics and the balance of the execution are correctly computed automatically once the algorithm ends the execution.

To perform this first Data Model test we have programmed the next Python code that, after executing, should store a row in the EUR-USD Opened Orders Table with the following fields:

- Open Price: 1.5
- Volume: 87
- Open Date Time: 2019-06-01 10:00:00
- Agent Mode: short
- Threshold Up: 0.005
- Threshold Down: 0.00375
- Open Event: upOvershoot
- Take Profit:2.8
- Stop Loss: 0.2
- Liquidity: 0.4
- Inventory: 450

After executing the code, we will verify if the order was successfully inserted in the database.

The result of the first test is shown in the figure below.

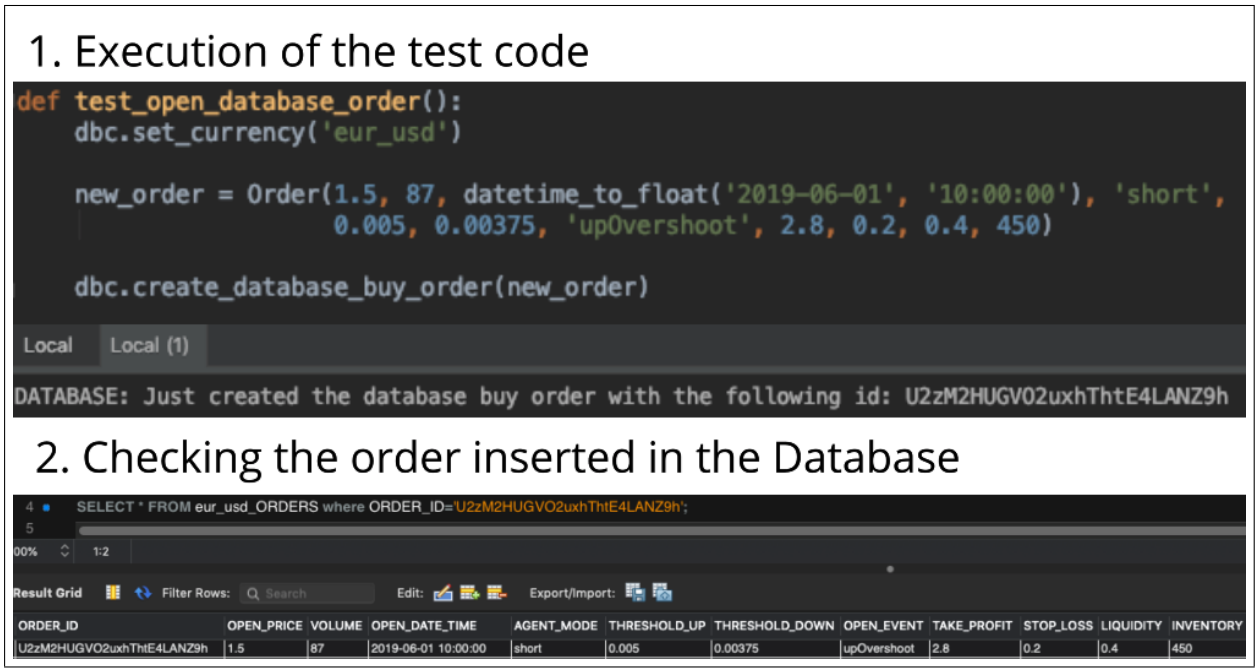


Figure 5.3: First test code and evidences that the row was properly inserted.

Afterwards, we close the order previously opened and we verify in the Database that the order was removed form the Opened Orders table and that it was stored in the Closed Orders table.

- Original Id: test_order_db
- Close Price: 2.8
- Close Date Time: 2019-06-05 14:30:00
- Close Event: directionalChangeToDown
- Close Option: TakeProfit

The result of the second test is shown in the figure below.

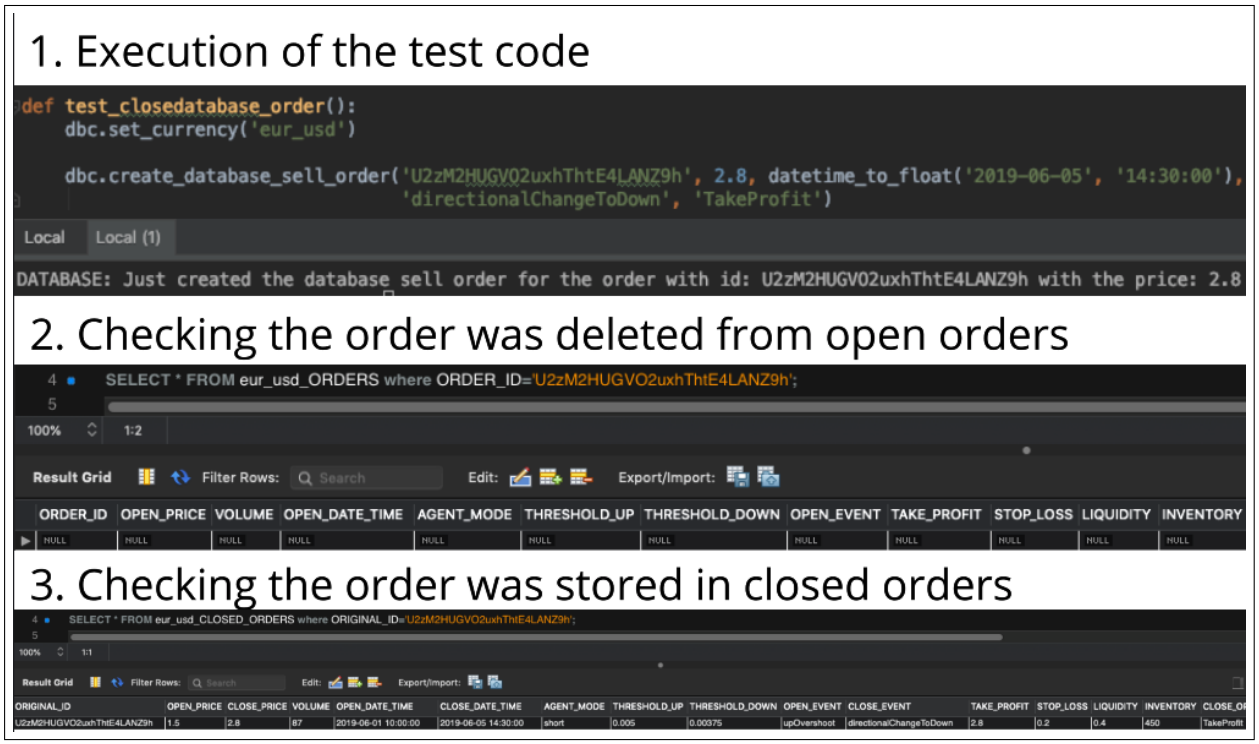


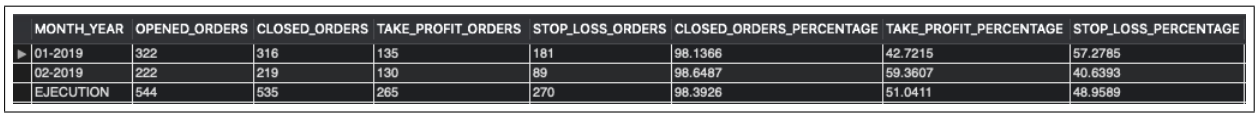
Figure 5.4: Second test code and evidences that the row was properly inserted as a closed order.

Lastly, we will perform an execution to verify that statistics data is consistent with the stored orders. The execution uses historical prices of the pair of currencies EUR/CHF from Janaury 2019 and Febraury 2019 with Original Threshold and Stop Loss value of 0.01%. The stats generated in a .csv file are shown below:

Month	Closed %	Take Profit %	Stop Loss %
01-2019	98.1366	42.7215	57.2785
02-2019	98.6487	59.3607	40.6393
EJECUTION	98.3926	51.0411	48.9589

Table 5.1: Stats of the orders managed by the algorithm in the last test.

Now we assert that the statistics match the data stored during the execution:



MONTH_YEAR	OPENED_ORDERS	CLOSED_ORDERS	TAKE_PROFIT_ORDERS	STOP_LOSS_ORDERS	CLOSED_ORDERS_PERCENTAGE	TAKE_PROFIT_PERCENTAGE	STOP_LOSS_PERCENTAGE
01-2019	322	316	135	181	98.1366	42.7215	57.2785
02-2019	222	219	130	89	98.6487	59.3607	40.6393
EJECUTION	544	535	265	270	98.3926	51.0411	48.9589

Figure 5.5: Monthly statistics stored in the Database.

As we can see, all the Datamodel test have been passed.

5.3 Integration Testing

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group.

Once we have tested individually the algorithm logic components by performing unitary tests and the database connection, in this section we are going to prove that all the modules interact correctly by carrying out an execution in real time. That is, we will simulate we are a regular user of the program, connecting the algorithm to an account of the trading platform in real time FXCM and we will verify that the algorithm detects intrinsic events and manages orders. Afterwards, we will check in the FXCM interface that the orders were opened and/or closed.

The result of the integration test is shown below. As the reader can see, the algorithm is able to open orders in the broker online platform.

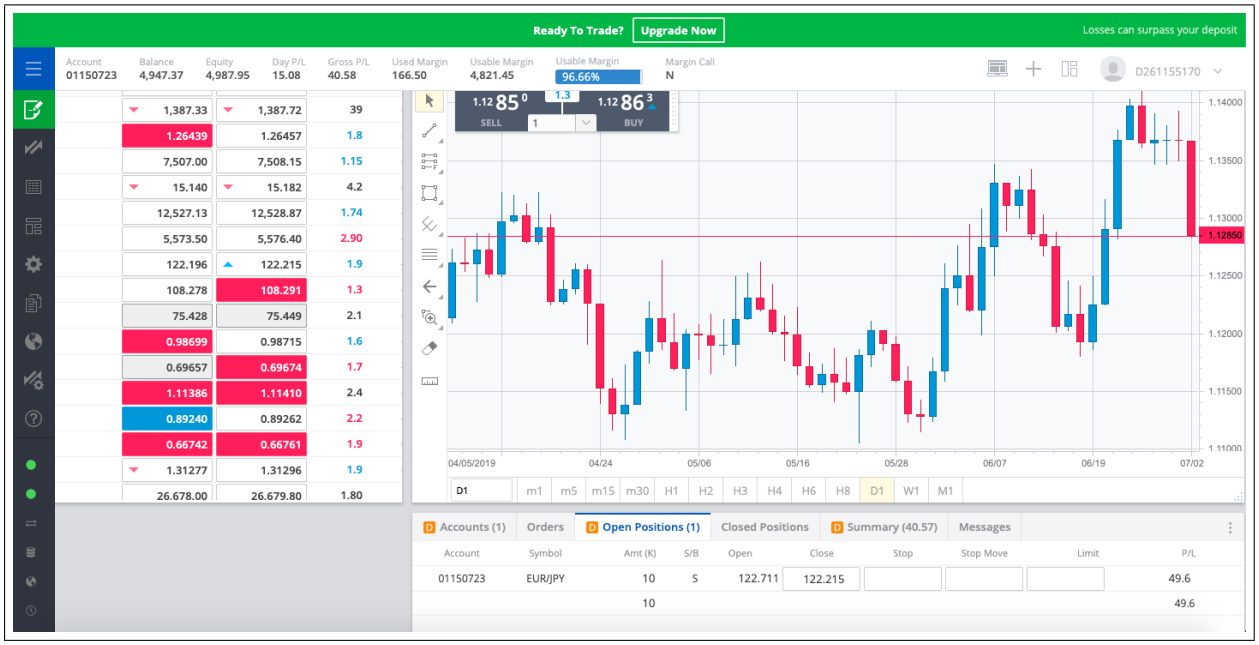


Figure 5.6: Checking that the algorithm open orders and they appear in the broker platform's interface.

5.4 Efficiency Testing

The fourth part of the testing corresponds to the tests of the efficiency between different versions of the algorithm.

Let us remind that, during Chapter 3, we have explained the complete development of the algorithm step by step. However, from the section we developed the Coastline Trading strategy, the algorithm could be used without further development. The additional develop-

ments were added to the aim of improving the benefits a user can obtain from the algorithm. Due to this reason, during this part we will perform efficiency tests on the returns of each of the versions of the algorithm.

The versions to be tested are the listed below:

- Version 1: version that implements the basic elements of the algorithm: detection of intrinsic events and Coastline Trading strategy.
- Version 2: version that contains the elements of the Version 1, added the Liquidity Indicator $\mathcal{L}$ used to control the aggressiveness with which orders are opened.
- Version 3: version that contains the elements of the Version 2, added asymmetric thresholds and the Inventory concept used to control the market trends. The features of this version matches with The Alpha Engine features
- Version 4: final version that contains the elements of the Version 3, added a risk management layer that sets Stop-Loss and Take-Profit prices every time an order is opened.

To perform these tests, we will compare the results of the balance and statistics of opened and closed orders in executions for each of the versions using the same data set, the same thresholds and the same number and mode of agents.

To analyze the results, two indicators will be used:

- Cumulative Liquid value: the cumulative liquid value is the total worth of the whole execution up to a certain time. It is computed by adding the total profit or loss of all the previously closed orders and the current situation on the opened orders not closed yet. This indicator is the most accurate to perform these efficiency tests, since it does not take into account only the closed orders, but also the opened at a certain point. So that, if the algorithm cannot to close a large number of orders that accumulate losses it will be penalized.
- Closed Rate: using the execution statistics, it will be analyzed the number of orders that, among all the opened, were succesfully closed.

Below, it is shown pseudocode of the function used to get the Cumulative Liquid Value in a certain point of the execution.

Algorithm 16 Function that computes the Cumulative Liquid Value

```
1:  function GET CUMULATIVE LIQUID VALUE()
2:
3:      cumulativeValue = 0
4:
5:      for each Order in CloseOrders do
6:
7:          if Order.mode is Long then
8:              cumulativeValue += Order.Volume * (Order.ClosePrice - Order.OpenPrice)
9:
10:         else if Order.mode is Short then
11:             cumulativeValue += Order.Volume * (Order.ClosePrice - Order.OpenPrice)
12:
13:     for each Order in OpenedOrders do
14:
15:         if Order.mode is Long then
16:             cumulativeValue += Order.Volume * (currentMarketPrice - Order.OpenPrice)
17:
18:         else if Order.mode is Short then
19:             cumulativeValue += Order.Volume * (Order.OpenPrice - currentMarketPrice)
20:
21:     return cumulativeValue
```

The environment chosen to perform this efficiency tests corresponds to the same published in the Alpha Engine code [23], so we can carry out a suitable comparison between their algorithm and ours and find out if we really have improved the efficiency.

It has been used:

- Four long agents and four short agents with the original threshold values 0.25%, 0.5%, 1% and 1.5%.
- Prices between Janaury 2006 and Janaury 2014.
- The pairs of currencies: AUD/USD, EUR/CHF, EUR/GBP, EUR/JPY, EUR/USD, GBP/USD, USD/CAD, USD/JPY, CHF/JPY, EUR/CAD, GBP/CHF, GBP/JPY, NZD/JPY, NZD/USD, USD/CHF, XAG/USD, XAU/USD

The results of the efficiency test performed on the differents algorithm versions are shown in the next subsections.

5.4.1 Version 1 Results

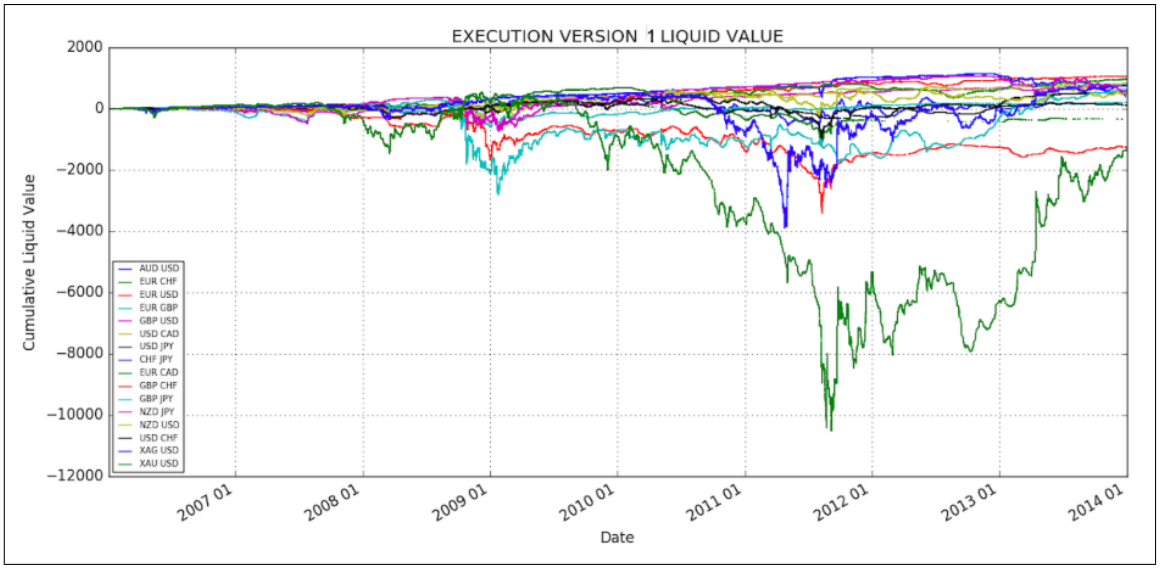


Figure 5.7: Graphic of the results of the Cumulative Liquid Values of the execution of the Version 1 in all of the aforementioned pairs of currencies

Currency	Opened	Closed	Closed Rate %
AUD/USD	31884	31138	97,6602
EUR/CHF	7809	7235	92,6495
EUR/GBP	10983	10483	95,4475
EUR/JPY	29144	28508	97,8177
EUR/USD	15939	15534	97,4590
GBP/USD	15956	15444	96,7911
USD/CAD	17275	16790	97,1924
USD/JPY	18561	17961	96,7674
CHF/JPY	25891	25372	97,9954
EUR/CAD	16730	16233	97,0292
GBP/CHF	16621	15776	94,9160
GBP/JPY	31430	30592	97,3337
NZD/JPY	53313	52287	98,0755
NZD/USD	35402	34642	97,8532
USD/CHF	17877	17182	96,1123
XAG/USD	135506	133695	98,6635
XAU/USD	53913	52326	97,0563
MEAN	31426	30659	97,5593

Table 5.2: Table of the Closed Rate for all the currencies in the Version 1

5.4.2 Version 2 Results

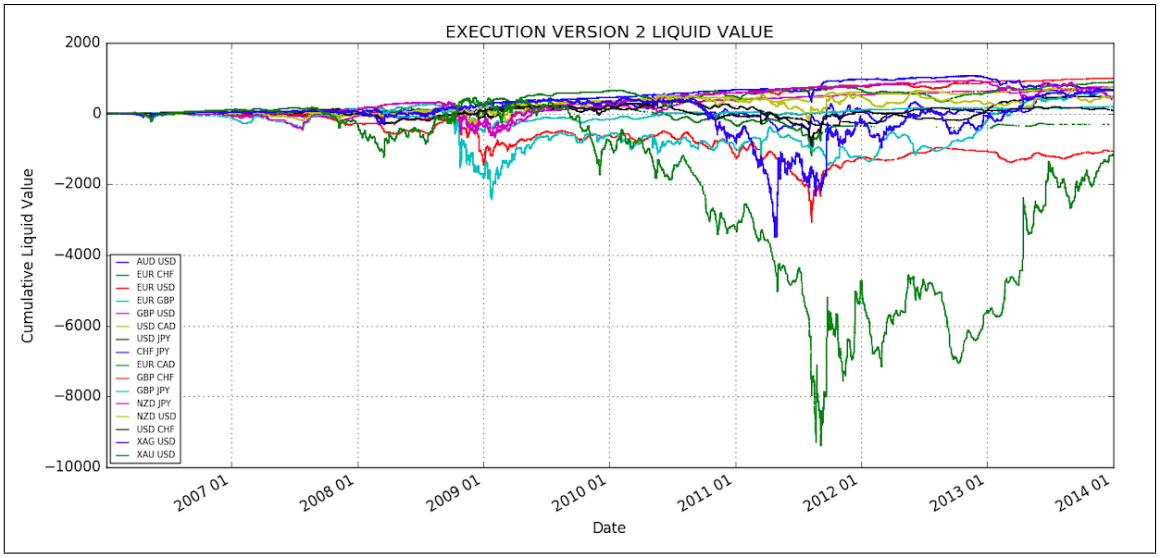


Figure 5.8: Graphic of the results of the Cumulative Liquid Values of the execution of the Version 2 in all of the aforementioned pairs of currencies

Currency	Opened	Closed	Closed Rate %
AUD/USD	31884	31138	97,6602
EUR/CHF	7809	7235	92,6495
EUR/GBP	10983	10483	95,4475
EUR/JPY	29144	28508	97,8177
EUR/USD	15939	15534	97,4590
GBP/USD	15956	15444	96,7911
USD/CAD	17275	16790	97,1924
USD/JPY	18561	17961	96,7674
CHF/JPY	25891	25372	97,9954
EUR/CAD	16730	16233	97,0292
GBP/CHF	16621	15776	94,9160
GBP/JPY	31430	30592	97,3337
NZD/JPY	53313	52287	98,0755
NZD/USD	35402	34642	97,8532
USD/CHF	17877	17182	96,1123
XAG/USD	135506	133695	98,6635
XAU/USD	53913	52326	97,0563
MEAN	31426	30659	97,5593

Table 5.3: Table of the Closed Rate for all the currencies in the Version 2

5.4.3 Version 3 Results

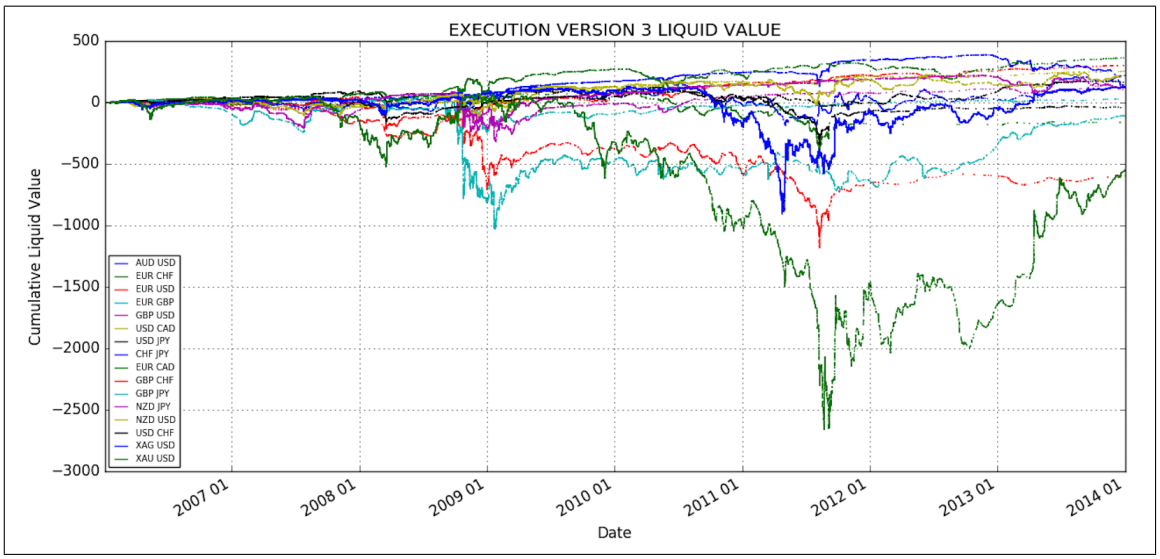


Figure 5.9: Graphic of the results of the Cumulative Liquid Values of the execution of the Version 3 in all of the aforementioned pairs of currencies

Currency	Opened	Closed	Closed Rate %
AUD/USD	17305	16979	98,1161
EUR/CHF	4883	4619	94,5934
EUR/GBP	6064	5826	96,0751
EUR/JPY	16039	15735	98,1046
EUR/USD	8319	8101	97,3794
GBP/USD	8564	8299	96,9056
USD/CAD	9208	8980	97,5238
USD/JPY	10727	10459	97,5016
CHF/JPY	13964	13749	98,4603
EUR/CAD	9467	9224	97,4331
GBP/CHF	9354	9021	96,440
GBP/JPY	16504	16136	97,7702
NZD/JPY	26122	25771	98,6563
NZD/USD	18720	18441	98,5096
USD/CHF	9543	9246	96,8877
XAG/USD	67746	67204	99,1999
XAU/USD	28242	27734	98,2012
MEAN	16516	16207	98,1291

Table 5.4: Table of the Closed Rate for all the currencies in the Version 3

5.4.4 Version 4 Results

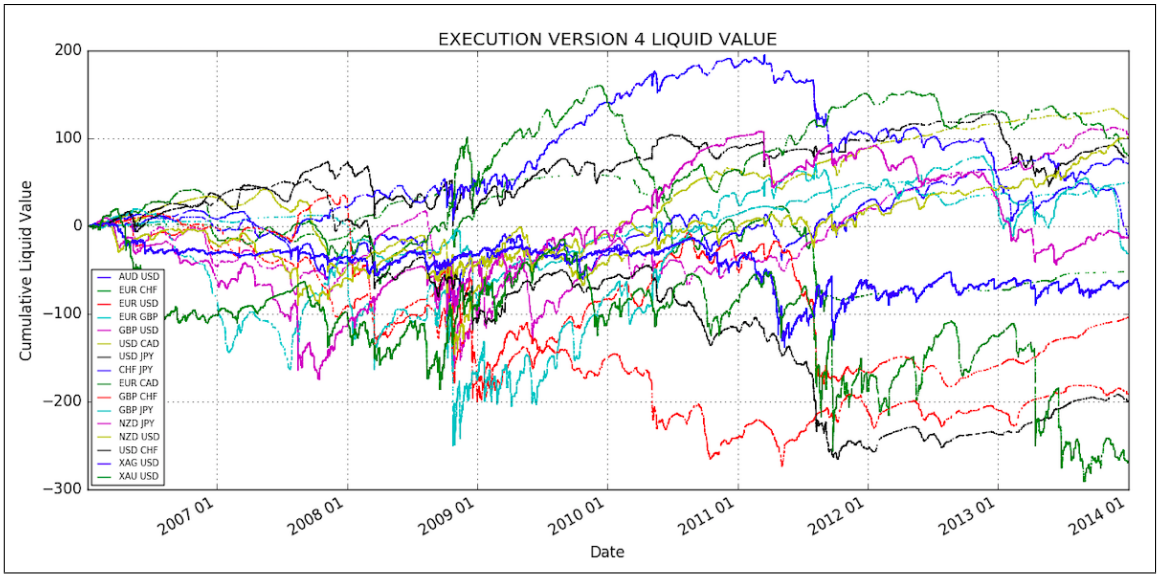


Figure 5.10: Graphic of the results of the Cumulative Liquid Values of the execution of the Version 4 in all of the aforementioned pairs of currencies

Currency	Opened	Closed	Closed Rate %
AUD/USD	25116	25017	99,6058
EUR/CHF	19890	19798	99,5374
EUR/GBP	8722	8608	98,6929
EUR/JPY	22760	22672	99,6133
EUR/USD	12536	12413	99,0188
GBP/USD	12322	12213	99,1154
USD/CAD	13236	13130	99,1991
USD/JPY	14290	14206	99,4121
CHF/JPY	19890	19798	99,5374
EUR/CAD	12941	12847	99,2736
GBP/CHF	12977	12862	99,1138
GBP/JPY	24559	24474	99,6538
NZD/JPY	42277	42187	99,7871
NZD/USD	27584	27468	99,5794
USD/CHF	13992	13886	99,2424
XAG/USD	114081	113988	99,9184
XAU/USD	43297	43204	99,7852
MEAN	25910	25810	99,6140

Table 5.5: Table of the Closed Rate for all the currencies in the Version 4

5.4.5 Understanding the results of the Efficiency Testing

Due to the large amount of documentation generated in this testing phase, the results shown in the graphs of the Cumulative Liquid Value show the results of the executions for all the currencies using the same version. As a reader that only uses these graphs, might not be able to see the detail in each of these graphs, so that, more detailed graphs of the executions in each of the currencies individually can be consulted in the Appendix attached.

As the reader can note from the graphs of the executions, the results are not uniform and vary a lot depending on the chosen currency pair. Although, in the general case, most of the currencies may present long-term gains, the currencies that generate losses represent a huge setback for the algorithm, because these losses are much more notable than the profits.

The currency with the most volatility is the pair XAG/USD. This market indicates the exchange price between Spot Silver and the US dollar. Although, this market is not in fact a real Forex currency market, we have used it in the executions because its large variations of values can serve to test how the different versions of the algorithm behave in situations of great market uncertainty. In this context, we see how our Version 4 is able to control the risk involved, that is, the abrupt changes in market prices, way better than the other versions since the price range of the Cumulative Liquid Value varies as follows:

- In Version 1, there is a loss up to -12,000 units at the end of 2011

- In Version 2, there is a loss up to -10,000 units at the end of 2011.
- In Version 3, there is a loss up to -3,000 units at the end of 2011
- In Version 4, during this risky end of 2011 period, there is a loss up to -250 units.

Although in versions 1, 2 and 3 after 2011 the Cumulative Liquid Value is stabilized, if the execution had been completed by the end of 2011, Version 4 would be the only one that could face the losses derived from the currency pair XAG/USD. As a real user of a trading system does not have umlimited money and time to invest and wait for the market to stabilize, we consider this point a huge factor in favor of the Version 4.

On the other hand, regarding the Close Rate, although the difference between versions may seem tiny, a 1% of the large number of orders managed by the algorithm throughout the execution can make a huge difference, since, generally, a losing order supposes way more risk than the reward that a winning order does. That is, it is very important that the algorithm manages to close as many of the orders as possible with profit, but in case profit cannot be achieved, the orders must be closed quickly before the losses accumulate.

The results of the Close Rate indicator obtained suggest that, although the percentage of profit closed order may be lower than the rest of the versions due to the Stop-Loss factor, again Version 4 seems the most appropriate to use in a real trading system, because it reduces the risk derived from large losses.

Once again, we address the reader to the appendix if he wants to consult the complete results in detail and judge the results by themselves.

To sum up and as a conclusion, the Version 4, which introduces a risk management layer to the algorithm, provides a gain in security to an investor who is investing real money and does not have unlimited time and money. As a proof of concept, Version 2 and Version 3 of the algorithm can meet and even obtain profits by removing the illiquid pairs which obtains no benefits, but if we had to choose a version to implement in a real system, we consider that, due to the risk control system introduced, Version 4 is the most appropriate.

Chapter 6

Environmental and Social impacts and Ethical and Professional Responsibilities

The environmental or anthropic impact is the effect produced by human activity on the environment. In this context, the impact of the project is the possible environmental impact that, for better or for worse, the development, implementation, testing and execution may cause. As long as the result of this Final Degree Project is a piece of software product, which does not require any manufacturing process or external component, beyond a computer to be run, we consider that the environmental impact of the project is negligible. However, it is worth mentioning that in order to put into operation the developed product, it will be required a functioning computer and an 24 hours on-line server containing the historical database so that the user is able to run tests historical when he wishes.

If the user chooses to connect the algorithm to the trading real time broker platform, the program must be kept running during all the entire execution time, so, consequently, the system needs to have a permanent source of energy, with a subsequent energy expenditure. This point, derived from the chosen Client-Server architectural pattern, might be solved in the future, by changing the architecture to a Cloud-based one. This change would improve both performance and the energy saving.

On the other hand, the social impact is the effect the product have on the well-being of the community. In this context, we consider that the social impact of the algorithm is greater than the environmental impact, due to the sensitive subject the trading world is. On one side, the positive social impact is that, due to its nature, the algorithm provides liquidity to the market. Since its emerge, the trading algorithm have been criticized and limited because its appearance used to affect negatively to the market micro-structure. The trading strategy used in our algorithm, the Coastline Trading, is a counter-trending strategy: the orders are opened when the rest of the market actors are selling, and these opened orders are sold when the rest of the market actors are buying.

Another social impact of the software product that can be taken into account is the fact that it allows to automate tasks that traditionally have been carried out by humans such as

analysing the market behaviour and the recent trends or managing the orders. Although this algorithm may not be yet a reliable version that allows to fully remove the human factor, it can be taken as a basis for future research to gradually erase the role of the human of the trading world.

Most of the Ethics and Professional responsibilities of the project is related to the process of obtaining, processing and storing the data used. Firstly, it is critical to ensure that the user personal account data is not stored, processed or used for the own behalf by the system developers. This personal data may contain sensitive economic data such as the monthly income. Additionally, the historical market data used to test and document the algorithm must be true, exact, complete and verifiable, so this data must not be rigged in order to show the most convenient results. The used historical prices data are public and have been obtained free of charge.

Regarding ethical responsibility, the authors of the Final Degree Project in no case intend to cause harm or loss to the user. However, in case the user wishes to invest real money, he must bear in mind that the algorithm does not guarantee at any point a winning formula to return income. It must be remembered that the algorithm is a proof of concept and, although certain simulations show profits, to perform these simulations the algorithm is supposed to have an unlimited amount of money to invest. If the user wants to perform a real investment, the amounts used to open orders should be adjusted according to the available money. For this reason, we do not recommend a user to invest a large amount.

Chapter 7

Conclusions, Model Criticism and Future Work

To conclude this project, in this last chapter we will take a look back and review all of the work that was done. This section will expose the general and personal conclusions that we have drawn from the study and subsequent development and implementation of the project. Finally, We will discuss multiple lines that may serve as a basis for future research and improvements to the work presented here.

7.1 Model Criticism

Throughout this project, a complete functional system capable of automating trading actions such as price analysis and order management has been studied, developed, and implemented.

This system's core is in charge of receiving market prices, analyzing them and determining before each of these new prices the action that has to be carried out. The algorithm can be used either to perform tests using a database of historical prices, or to trade live connecting to a broker platform. In much of the development, we have relied on research carried out over several decades of study, which culminated in a proof-of-concept algorithm called "The Alpha Engine".

From the beginning, our main objective has been to develop a similar system to The Alpha Engine that could be used in the real world by any user, and to introduce elements that this model lacks in order to improve its efficiency and operation.

As the reader may have guessed, the main challenge in the development of the system has been mainly the study and implementation of the logical components of the system. These components come from numerous statistical and mathematical investigations carried out over decades of study. It is worth mentioning that, thanks to these investigations, it has been possible to study, justify and dissect the DNA of the developed algorithm.

After having taken into account and practically used as a base and development guide the algorithm presented by Golub et al. [22] in its paper "The Alpha Engine", we have also reached some conclusions related to the theoretical basis and the implementation that this

algorithm uses, and we would like to take this point to present our critical vision of the line of research referred to throughout the Development chapter.

First, it seems inappropriate to build a trading algorithm without taking into account money or time. In this context we must note that, like the authors of The Alpha Engine, the models up to this moment are a proof of concept with the aim of noting that the investigations of the directional-change event approach to model trading systems can provide results if the investigation continues. But, from now on, we believe that future researchers should take into account that the trader must have a maximum budget with which to invest in the market.

Another weak aspect that we have found is that the documentation referring to the liquidity indicator is not entirely clear. The liquidity indicator was by far the hardest module of the entire system to develop, understand and explain. To develop it, we used several documents that are referred to in this document's bibliography.

However, these documents do not clearly indicate the methodology and development of this module. The code published in GitHub by the authors of The Alpha Engine is neither well documented nor self-explanatory. There are multiple variables and methods whose names are not descriptive, and there are variables that have values chosen arbitrarily without any justification. We think that the published code could have been better documented, better structured, could have had more descriptive names and followed better code patterns and recommendations.

Finally, we would like to mention that, in our opinion, our version of the algorithm can improve the performance in a real execution with respect to previously developed algorithms based in directional-change events, which was one of our main objectives at the beginning of the of this project. The aspects that we think we managed to improve with respect to The Alpha Engine are the following:

- The fact of providing an architecture to the system that allows a user to easily interact with the algorithm and see for themselves the results of the executions.

- Up to this point, the Alpha Engine only allowed testing the model but did not provide a data feed, which forced the user to obtain its own data feed. Our system allows the user to use the algorithm in the real world when connected to a trading platform. In addition, the user will not need to provide their own data feed, the algorithm can connect to our historical database which will pass the data directly to the algorithm.

- As the performance tests have showed us, thanks to the risk management system developed, our algorithm can independently manage money from an investor in a much safer way compared.

- Therefore, a user who trades with limited time and money, will obtain obtain acceptable results in the medium and long term due to the increased performance of our algorithm.

7.2 Personal and educational conclusions

Continuing with the development and implementation of the project, several methodologies have been followed, depending on the functionality of the system that was intended to be implemented:

- Documentation and study based on existing research. This is the case of the developments of the algorithm modules that had been previously developed and whose development is detailed in the documents taken as reference throughout the project. This is the case of the detection of intrinsic events, the Coastline Trading strategy, the liquidity indicator and the asymmetric thresholds for the detection of events.
- Own investigation, trial and error. Specifically, in the case of finding the best risk control method, we had to investigate and choose between the different strategies used in the real trading world. Once the strategy was chosen, we had to identify the optimal values with which to assign the output values of the opened orders. We executed several simulations for multiple values and we discovered that the algorithm is very sensitive to the market and the threshold value, this helped us to later determine a function that assigns risk to an order depending on the market liquidity and the threshold value.
- Application of knowledge acquired during our software degree. For the implementation and testing phases of the system, we applied knowledge obtained in various courses throughout the degree. For instance, defining the requirements and architecture or unit tests of the system are part of the software construction process that we have learned.

On the other hand, in the personal field, we feel that we have grown in aspects that every engineer must have. We have increased our resilience and improved our self-learning ability.

The realization of this project has allowed us to obtain skills that when we started, we did not think we were going to learn in this process. The fact that we had to soak up a multitude of scientific documents, in large part with advanced statistical and mathematical concepts, has encouraged us to know that in the future we will be able to face projects with concepts that we ignore and that can be intimidating at the beginning.

In addition, most of the technologies used in the final system were unknown to us before starting the project. We would also like to name many other aspects that, although we have not gone into much depth, are present in the project. Big Data in the mining, storage and processing of data, the creation of a client-server architecture system, development of a graphical user interface, unit tests, the automation of efficiency tests and the analysis of the data produced by the system.

Regarding aspects not related with technology, we highlight the theoretical knowledge acquired about macroeconomics and about the structure of the stock and currency markets. We now have a starting point in a topic of our interest, as it is algorithmic trading, to continue studying and developing better future projects.

In summary, this project has helped us to grow enormously. For us it has been an experience that we could have hardly ever experienced doing another type of project.

7.3 Line for future research

Lastly, and in order to put an end to this project, we would like to mention several general aspects of the project that in our opinion can be improved and hopefully can serve as the basis of study for future research.

As the main weakness of the algorithm, we know that the chosen architecture is not the most optimal. The choice of a Client-Server architecture implies that the transactions between the database and the algorithm does not have the highest performance.

Given that a historical execution of our algorithm is executed using several agents and the whole dataset available for a market can get to deal with data of up to 7 million market prices and can open up to 500,000 orders, improving performance in transactions between the algorithm and the database is a critical point. An architecture in which the database and the logic of the algorithm meet the server itself, such as in the cloud, could suppose a great improvement in the efficiency of these transactions, and, therefore, in the time it takes the algorithm in carrying out these executions, with the consequent energy saving that this architecture can suppose.

Another aspect to keep in mind is that the current algorithm is very sensitive to the threshold value that is chosen. A line of future research could be to perform multiple executions with different thresholds to detect what is the optimal threshold value in each of the currencies and, if applicable, determine which of the currencies are not good for our strategy.

Finally, the risk control system, which is based on the Risk / Reward Ratio principle and which makes use of the threshold and liquidity values to determine the values of the Take-Profit and the Stop-Loss of an action at the time of being opened can be optimized in the future. In this line, more methods and values of Stop-Loss could be executed to determine a system that is able to reduce the risk that the algorithm takes in an optimal way.

Source Code

```
import tkinter as tk
from tkinter import ttk
import classesBroker as alpha
import time
import matplotlib
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
import matplotlib.pyplot as plt
import fxcmpy
import matplotlib.dates as mdates
import DBConnection as dbc
from historicClasses import Agent
from historicClasses import Price
import threading

matplotlib.use("TkAgg")

LARGE_FONT = ('Verdana', 20)

currency = '',
start_year = 0
end_year = 0
token = ''
threshold = 0.01
agents = []
historic_text = ''
historic_title = ''
cont = 0
data = None
live = True
con = None


def trade():
```

```
    global currency, agents, live

    initiate_traders()

    while live:
        time.sleep(3)

        try:
            last_price = con.get_last_price(currency)
            for agent in agents:
                agent.trade(last_price)
        except:
            time.sleep(10)
            con.close()
            initiate_traders()


def initiate_traders():
    global agents, con, currency

    con = fxcmpy.fxcmpy(access_token=token, log_level='error')
    con.subscribe_market_data(currency)
    agent_long = alpha.Agent(con, currency, 0.001, agent_mode='
        long')
    agent_short = alpha.Agent(con, currency, 0.001, agent_mode='
        short')
    agent_long2 = alpha.Agent(con, currency, 0.0025, agent_mode='
        long')
    agent_short2 = alpha.Agent(con, currency, 0.0025, agent_mode='
        short')

    agents = [agent_long, agent_short, agent_long2, agent_short2]


def run_historic():

    global currency, start_year, end_year, threshold,
        historic_text, historic_title, data

    dataset = dbc.get_dataset(currency, start_year, end_year)
    dbc.reset_database()
    time.sleep(1)

    agent_long = Agent(original_threshold=threshold, agent_mode='
```

```
        long')
    agent_short = Agent(original_threshold=threshold, agent_mode='
        short')

    for row in dataset:
        id = row[0]
        datetime = dbc.datetime_to_float(str(row[2]), str(row[3]))
        bid = float(row[5])
        ask = float(row[6])
        price = Price(id, ask, bid, datetime)
        agent_long.trade(price)
        agent_short.trade(price)

    dbc.get_total_stats()
    stats_data = dbc.print_stats()

    data = dbc.get_liquid_value()

    time.sleep(1)

    historic_text = \
        f'Opened orders: {stats_data.loc[0, "OPENED_ORDERS"]}\n\n' \
        f'Closed orders: {stats_data.loc[0, "CLOSED_ORDERS"]}\n\n' \
        f'Orders closed with profit: {stats_data.loc[0, "
            TAKE_PROFIT_ORDERS"]}\n\n' \
        f'Orders closed with stop loss: {stats_data.loc[0, "
            STOP_LOSS_ORDERS"]}\n\n' \
        f'Percentage of closed orders: {stats_data.loc[0, "
            CLOSED_ORDERS_PERCENTAGE"]} %\n\n' \
        f'Percentage of orders closed with profit: {stats_data.loc
            [0, "TAKE_PROFIT_PERCENTAGE"]} %\n\n' \
        f'Percentage of orders closed with stop loss: {stats_data.
            loc[0, "STOP_LOSS_PERCENTAGE"]} %\n\n'

    historic_title = f'{currency} results between {start_year} and
        {end_year}\n\n'
    print(0)


class Main(tk.Tk):

    def __init__(self, *args, **kwargs):
```

```
        tk.Tk.__init__(self, *args, **kwargs)

        tk.Tk.wm_title(self, 'AlphaPy')

        container = tk.Frame(self)
        container.pack(side="top", fill='both', expand=True)

        container.grid_rowconfigure(0, weight=1)
        container.grid_columnconfigure(0, weight=1)

        self.frames = {}

        for F in (StartPage, LiveStart, HistoricStart, Live,
           Historic, GraphPage):
            frame = F(container, self)
            self.frames[F] = frame
            frame.grid(row=0, column=0, sticky="nsew")

        self.show_frame(StartPage)

    def show_frame(self, cont):
        frame = self.frames[cont]
        frame.tkraise()

    def stop_live(self, cont):
        global con, live

        con.close()
        live = False

        frame = self.frames[cont]
        frame.tkraise()

    def change_to_live(self, cont, token_var, currency_var):
        global token, currency, live

        live = True
        token = token_var.get()
        currency_aux = currency_var.curselection()
        currency = currency_var.get(currency_aux)

        x = threading.Thread(target=trade)
        x.start()
```

```
        frame = self.frames[cont]
        frame.tkraise()

    def change_to_historic(self, cont, start, end, threshold_var,
        currency_var):
        global threshold, start_year, end_year, currency

        start_year = int(start.get())
        end_year = int(end.get())
        threshold = float(threshold_var.get())

        currency_aux = currency_var.curselection()
        currency = currency_var.get(currency_aux)

        run_historic()

        frame = self.frames[cont]
        frame.title.config(text=historic_title)
        frame.text.config(text=historic_text)
        frame.tkraise()

    def change_to_graph(self, cont):

        global data

        liquid_value = data['BALANCE'].tolist()
        dates = data['DATE'].tolist()

        frame = self.frames[cont]
        frame.a.clear()
        frame.a.set_title(f'Results of {currency} with threshold {
            threshold}\n\n')
        frame.a.plot(dates, liquid_value)
        frame.canvas.draw()
        frame.tkraise()


class StartPage(tk.Frame):

    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)

        label = tk.Label(self, text='Welcome!!!', font=LARGE_FONT,
            width="20", height="3")
```

```
        label.pack(pady=70, padx=250)

        live_button = ttk.Button(self, text="Live Trading",
            command=lambda: controller.show_frame(LiveStart))
        live_button.pack(pady=50)

        historic_button = ttk.Button(self, text="Historic Trading
            ",
                                     command=lambda: controller.
                                         show_frame(HistoricStart))
        historic_button.pack(pady=50)

        blank_space = tk.Label(self, text="", font=LARGE_FONT,
            width="20", height="3")
        blank_space.pack(pady=20)


class LiveStart(tk.Frame):

    global token

    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)

        blank_space = tk.Label(self, text="", font=LARGE_FONT)
        blank_space.pack(pady=40, padx=400)

        label = tk.Label(self, text="Insert your token", font=17)
        label.pack(pady=10, padx=10)

        token_var = tk.StringVar()

        token_entry = ttk.Entry(self, width=30, textvariable=
            token_var)
        token_entry.pack(pady=10, padx=10)

        our_token = tk.Label(self, text="If you don't have a token
            , you can use ours: \n"
        " 21c5c79ea3fa48989a892a9e496a0bc7d4cf9480", font=14)
        our_token.pack(pady=5, padx=10)

        blank_space2 = tk.Label(self, text="", font=LARGE_FONT)
        blank_space2.pack(pady=20)
```

```
        label = tk.Label(self, text="Select your currency", font
            =17)
        label.pack(pady=10, padx=10)

        currencies = tk.StringVar(value=['EUR/USD', 'GBP/USD', '
            USD/JPY', 'EUR/JPY', 'USD/EUR', 'BTC/USD'])
        currency_entry = tk.Listbox(self, listvariable=currencies,
             height=6)
        currency_entry.pack()

        blank_space3 = tk.Label(self, text="", font=LARGE_FONT)
        blank_space3.pack(pady=20)

        start_button = ttk.Button(self, text="Start", command=
            lambda: controller.change_to_live(Live, token_var,




        start_button.pack(pady=10, padx=10)

        back_button = ttk.Button(self, text="BACK", command=lambda
            : controller.show_frame(StartPage))
        back_button.pack(pady=10, padx=10)


class HistoricStart(tk.Frame):

    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)

        blank_space0 = tk.Label(self, text="", font=LARGE_FONT)
        blank_space0.pack(pady=5)

        label = tk.Label(self, text="Select start year", font=17)
        label.pack(pady=5)

        start_date = ttk.Combobox(self, values=('2001', '2002',
            '2003', '2004', '2005', '2006', '2007', '2008',
                                                '2009', '2010',
                                                   '2011', '2012',
                                                    '2013',
                                                   '2014', '2015',
                                                    '2016',
```

```
                                                    '2017', '2018',
                                                      '2019'))
        start_date.pack()

        label = tk.Label(self, text="Select end year", font=17)
        label.pack(pady=10)

        finish_date = ttk.Combobox(self, values=('2001', '2002',
            '2003', '2004', '2005', '2006', '2007', '2008',
                                                    '2009', '2010',
                                                      '2011', '2012',
                                                       '2013',
                                                      '2014', '2015',
                                                       '2016',
                                                    '2017', '2018',
                                                      '2019'))
        finish_date.pack()

        blank_space1 = tk.Label(self, text="", font=LARGE_FONT)
        blank_space1.pack(pady=5)

        label = tk.Label(self, text="Select a threshold", font=17)
        label.pack(pady=7, padx=10)

        threshold_entry = ttk.Combobox(self, values=('0.01',
            '0.005', '0.0025', '0.0015'))
        threshold_entry.pack()

        blank_space2 = tk.Label(self, text="", font=LARGE_FONT)
        blank_space2.pack(pady=5)

        label = tk.Label(self, text="Select your currency", font
            =17)
        label.pack(pady=7, padx=10)

        currencies = tk.StringVar(value=['eur_usd', 'gbp_usd', '
            usd_cad', 'usd_jpy', 'eur_gbp', 'eur_jpy', 'eur_chf'])
        currency_entry = tk.Listbox(self, listvariable=currencies,
             height=7)
        currency_entry.pack()

        blank_space3 = tk.Label(self, text="", font=17)
        blank_space3.pack(pady=5)
```

```
        wait_label = tk.Label(self, text=f'This process may take a
            while,\n\n'
                                        f'Please be patient\n',
                                           font=17)
        wait_label.pack()

        start_button = ttk.Button(self, text="Start", command=
           lambda: controller.change_to_historic(Historic,
           start_date, finish_date, threshold_entry,
           currency_entry))
        start_button.pack(pady=10, padx=10)

        back_button = ttk.Button(self, text="BACK", command=lambda
           : controller.show_frame(StartPage))
        back_button.pack(pady=10, padx=10)

        blank_space4 = tk.Label(self, text="", font=17)
        blank_space4.pack(pady=10)


class Live(tk.Frame):

    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)

        blank_space0 = tk.Label(self, text="", font=LARGE_FONT)
        blank_space0.pack(pady=20)

        algo_status = tk.Label(self, text=f'The algorithm is now
           trading automatically', font=LARGE_FONT)
        algo_status.pack(pady=20)

        monitor_text = tk.Label(self, text=f'You can monitor your
           orders at https://tradingstation.fxcm.com', font=
           LARGE_FONT)
        monitor_text.pack(pady=20)

        user_info = tk.Label(self, text=f'If you are using our
           token, use this user and password to access the website
           : \n\n'
                                        f'User: D261146687\n'
                                        f'Password: 6651\n', font=
                                           LARGE_FONT)
        user_info.pack(pady=20)
```

```
        blank_space1 = tk.Label(self, text="", font=LARGE_FONT)
        blank_space1.pack(pady=20)

        back_button = ttk.Button(self, text="BACK", command=lambda
            : controller.stop_live(StartPage))
        back_button.pack(padx=200)


class Historic(tk.Frame):

    global start_year, end_year, historic_text, currency,
        historic_title

    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)

        blank_space0 = tk.Label(self, text="", font=LARGE_FONT)
        blank_space0.pack(pady=10)

        self.title = tk.Label(self, text=historic_title, font=
            LARGE_FONT)
        self.title.pack(pady=20, padx=10)

        self.text = tk.Label(self, text=historic_text, font=17)
        self.text.pack(pady=10, padx=10)

        graph_button = ttk.Button(self, text="Graph of the liquid
            value", command=lambda: controller.change_to_graph(
            GraphPage))
        graph_button.pack(pady=10, padx=10)

        back_button = ttk.Button(self, text="Home", command=lambda
            : controller.show_frame(StartPage))
        back_button.pack(pady=10, padx=10)


class GraphPage(tk.Frame):

    global start_year, end_year, historic_text, currency,
        historic_title, threshold

    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)
```

```
        blank_space0 = tk.Label(self, text="", font=LARGE_FONT)
        blank_space0.pack(pady=3)

        plt.style.use('seaborn')
        plt.legend(loc='best', prop={'size': 20})

        self.fig = Figure(figsize=(12, 5), dpi=100)

        self.a = self.fig.add_subplot(111)

        self.a.xaxis.set_major_locator(mdates.YearLocator())
        self.a.xaxis.set_major_formatter(mdates.DateFormatter('%Y
            %m'))

        self.canvas = FigureCanvasTkAgg(self.fig, self)
        self.canvas.get_tk_widget().pack(side=tk.BOTTOM, fill=tk.
            BOTH, expand=True)
        self.canvas._tkcanvas.pack(side=tk.TOP, fill=tk.BOTH,
            expand=True)

        back_button = ttk.Button(self, text="back", command=lambda
            : controller.show_frame(Historic))
        back_button.pack(pady=10, padx=10)


app = Main()
app.mainloop()
```

Class DBConnection

```
#!/usr/bin/python3
import pymysql as sql
import datetime as dt
import matplotlib as mpl
import pandas as pd

selected_currency = None

onlinehost = '160.153.142.193'
onlineuser = 'Forex_DB_TFG'
onlinepassword = 'UVx.P2Ps6V@L+.JL'
onlinedb = 'Forex_DB_TFG'
onlineport = 3306
```

```
balance = 0.0
opened_orders = 0
closed_orders = 0
take_profit_orders = 0
stop_loss_orders = 0
current_month = 1
current_year = 2001

db = sql.connect(host=onlinehost, user=onlineuser, password=
    onlinepassword, db=onlinedb, autocommit=True,
                 charset='utf8', port=onlineport)

end_year = 2019
start_year = 2001

mpl.rcParams.update({'figure.max_open_warning': 0})


def execute_sentence(sentence, cursor):
    """
    By calling this method, and using the PyMySQL library, you can
        execute a sentence in the connected database.
    :param sentence: the sentence you want to execute
    :return: the results of the sentence
    """

    try:
        if cursor.connection:
            cursor.execute(sentence)
            return cursor.fetchall()

        else:
            print("impossible to connect to the database")

    except Exception as e:
        return str(e)


def datetime_to_float(date, time):
    """
    Auxiliar method used to convert a date and a time into a
       floating time value (time value that isn't tied to a
       specific time zone)
```

```
    :param date: date of a row of the dataset. time: time of a row
        of the dataset
    :return: the datetime in floating format
    """

    year, month, day = date.split("-")
    hour, minute, second = time.split(":")
    date = dt.datetime(int(year), int(month), int(day), int(hour),
        int(minute), int(second))
    return date.timestamp()


def float_to_datetime(float):
    """
    Auxiliar method used to convert a floating datetime into a
       python datetime value
    :param float: floating datetime value
    :return: python datetime value
    """

    return str(dt.datetime.fromtimestamp(float))


def get_dataset(currency, start_y, end_y):
    global db
    global selected_currency
    global end_year
    global start_year
    global current_year

    reset_global_variables()

    selected_currency = currency
    assert selected_currency not in 'none', "{}, Not a valid
       currency selected".format(selected_currency)

    print("Select the dataset start year (min. 2001)")
    start_year = start_y
    assert (start_year >= 2001 and start_year <= 2019), "{}, Not a
        valid start year selected".format(start_year)
    current_year = start_year
    start_date = '{}-01-01'.format(start_year)

    print("Select the dataset end year (max. 2019)")
```

```
    end_year = end_y
    assert (end_year >= 2001 and start_year <= 2019), "{}, Not a
       valid start year selected".format(start_year)

    if end_year != 2019:
        end_date = '{}-12-31'.format(end_year)

    else:
        end_date = '{}-03-31'.format(end_year)

    get_dataset_sentence = "SELECT * FROM {} WHERE DATE_T BETWEEN
       \'{}\' AND \'{}\'".format(selected_currency,









    dataset = execute_sentence(get_dataset_sentence, db.cursor())
    db.cursor().close()

    return dataset


def create_database_buy_order(order):
    global db
    global current_month
    global opened_orders
    global selected_currency

    order_date = float_to_datetime(order.time)
    order_date = order_date.split("-")

    if (int(order_date[1]) != current_month):
        update_stats()

    opened_orders += 1

    insert_sentence = "INSERT INTO {}_ORDERS (ORDER_ID, OPEN_PRICE
       , VOLUME, OPEN_DATE_TIME, AGENT_MODE, THRESHOLD_UP,
       THRESHOLD_DOWN, OPEN_EVENT, TAKE_PROFIT, STOP_LOSS,
       LIQUIDITY, INVENTORY, CLOSED) " \
                        "VALUES (\'{}\', {}, {}, \'{}\', \'{}\', {},
```

```
                        {}, \'{}\', {}, {}, {}, {}, 0);".format(
            selected_currency,
            order.order_id, order.open_price,
            order.volume,
            float_to_datetime(order.time),
            order.agent_mode,
            order.threshold_up,
            order.threshold_down,
            order.event_of_creation,
            order.take_profit,
            order.stop_loss,
            order.liquidity,
            order.inventory)

    execute_sentence(insert_sentence, db.cursor())
    print("DATABASE: Just created the database buy order with the
       following id: {}".format(order.order_id))

    update_the_balance_buy(order.open_price, float_to_datetime(
       order.time))


def create_database_sell_order(id, sell_price, close_time,
   close_event_name, close_option):
    global db
    global closed_orders
    global take_profit_orders
    global stop_loss_orders
    global selected_currency

    order_date = float_to_datetime(close_time)
    order_date = order_date.split("-")

    if (int(order_date[1]) != current_month):

        update_stats()

    closed_orders += 1

    if (close_option == 'StopLoss'):
        stop_loss_orders += 1

    elif (close_option == 'TakeProfit'):
        take_profit_orders += 1
```

```
    select_sentence = "SELECT * FROM {}_ORDERS WHERE ORDER_ID =
        \'{}\';".format(selected_currency, id)
    delete_sentence = "DELETE FROM {}_ORDERS WHERE ORDER_ID =
        \'{}\';".format(selected_currency, id)

    original_order = execute_sentence(select_sentence, db.cursor()
        )
    execute_sentence(delete_sentence, db.cursor())

    for row in original_order:
        insert_sentence = "INSERT INTO {}_CLOSED_ORDERS (
            ORIGINAL_ID, OPEN_PRICE, CLOSE_PRICE, VOLUME,
            OPEN_DATE_TIME, CLOSE_DATE_TIME, AGENT_MODE,
            THRESHOLD_UP, THRESHOLD_DOWN, OPEN_EVENT, CLOSE_EVENT,
            TAKE_PROFIT, STOP_LOSS, LIQUIDITY, INVENTORY,
            CLOSE_OPTION) " \
                          "VALUES (\'{}\', {}, {}, {}, \'{}\',
                              \'{}\', \'{}\', {}, {}, \'{}\',
                              \'{}\', {}, {}, {}, {}, \'{}\');".
                              format(
            selected_currency, row[0], row[1], sell_price, row[2],
                row[3], float_to_datetime(close_time), row[4],
            row[5], row[6], row[7], close_event_name, row[8], row
                [9], row[10], row[11], close_option)
        execute_sentence(insert_sentence, db.cursor())

    select_order_inserted = "SELECT * FROM {}_CLOSED_ORDERS WHERE
        ORIGINAL_ID = \'{}\';".format(selected_currency, id)

    sell_order = execute_sentence(select_order_inserted, db.cursor
        ())

    update_the_balance_sell(sell_order, sell_price)


def update_stats():

    global selected_currency, opened_orders, closed_orders,
        take_profit_orders, stop_loss_orders, current_month,
        current_year


    if opened_orders != 0:
```

```
if closed_orders != 0:
    if current_month >= 1 and current_month <= 9:
        insert_sentence = """
                            INSERT INTO {}_MONTHLY_STATS (
                                MONTH_YEAR, OPENED_ORDERS,
                                CLOSED_ORDERS,
                                TAKE_PROFIT_ORDERS,
                                STOP_LOSS_ORDERS,
                                CLOSED_ORDERS_PERCENTAGE,
                                TAKE_PROFIT_PERCENTAGE,
                                STOP_LOSS_PERCENTAGE)
                            VALUES(\'0{}-{}\', {}, {}, {},
                                {}, {}, {}, {})
                            """.format(selected_currency,
                                current_month, current_year
                                , opened_orders,
                                closed_orders,
                                take_profit_orders,
                                stop_loss_orders,
                                closed_orders /
                                opened_orders * 100,
                                take_profit_orders/
                                closed_orders * 100,
                                stop_loss_orders /
                                closed_orders * 100)

    elif current_month >=10 and current_month<= 12:
        insert_sentence = """
                            INSERT INTO {}_MONTHLY_STATS (
                                MONTH_YEAR, OPENED_ORDERS,
                                CLOSED_ORDERS,
                                TAKE_PROFIT_ORDERS,
                                STOP_LOSS_ORDERS,
                                CLOSED_ORDERS_PERCENTAGE,
                                TAKE_PROFIT_PERCENTAGE,
                                STOP_LOSS_PERCENTAGE)
                            VALUES(\'{}-{}\', {}, {}, {},
                                {}, {}, {}, {})
                            """.format(selected_currency,
                                current_month, current_year
                                , opened_orders,
                                closed_orders,
                                take_profit_orders,
                                stop_loss_orders,
```

```
                                        closed_orders /
                                        opened_orders * 100,
                                        take_profit_orders/
                                        closed_orders * 100,
                                        stop_loss_orders /
                                        closed_orders * 100)
            execute_sentence(insert_sentence, db.cursor())

        elif closed_orders == 0:
            if current_month >= 1 and current_month <= 9:
                insert_sentence = """
                                    INSERT INTO {}_MONTHLY_STATS (
                                        MONTH_YEAR, OPENED_ORDERS,
                                        CLOSED_ORDERS,
                                        TAKE_PROFIT_ORDERS,
                                        STOP_LOSS_ORDERS,
                                        CLOSED_ORDERS_PERCENTAGE,
                                        TAKE_PROFIT_PERCENTAGE,
                                        STOP_LOSS_PERCENTAGE)
                                    VALUES(\'0{}-{}\', {}, 0, 0,
                                        0, 0, 0, 0)
                                    """.format(selected_currency,
                                        current_month, current_year
                                        , opened_orders)

            elif current_month >= 10 and current_month <= 12:
                insert_sentence = """
                                    INSERT INTO {}_MONTHLY_STATS (
                                        MONTH_YEAR, OPENED_ORDERS,
                                        CLOSED_ORDERS,
                                        TAKE_PROFIT_ORDERS,
                                        STOP_LOSS_ORDERS,
                                        CLOSED_ORDERS_PERCENTAGE,
                                        TAKE_PROFIT_PERCENTAGE,
                                        STOP_LOSS_PERCENTAGE)
                                    VALUES(\'{}-{}\', {}, 0, 0, 0,
                                         0, 0, 0)
                                    """.format(selected_currency,
                                        current_month, current_year
                                        , opened_orders)
            execute_sentence(insert_sentence, db.cursor())

    opened_orders = 0
    closed_orders = 0
```

```
    take_profit_orders = 0
    stop_loss_orders = 0

    if current_month == 12:
        current_month = 1
        current_year += 1
    else:
        current_month += 1


def update_the_balance_buy(current_price, date):
    global db, balance

    select_all = "SELECT * FROM {}_ORDERS;".format(
       selected_currency)

    opened_orders = execute_sentence(select_all, db.cursor())

    current = 0.0

    for row in opened_orders:
        if row[4] == 'long':
            current += (current_price * row[2] - row[1] * row[2])

        elif row[4] == 'short':
            current += (row[1] * row[2] - current_price * row[2])

    insert_balance(balance + current, date)


def update_the_balance_sell(sell_order, current_price):
    global db, balance

    select_all = "SELECT * FROM {}_ORDERS;".format(
       selected_currency)

    opened_orders = execute_sentence(select_all, db.cursor())

    current = 0.0

    for row in sell_order:
        if row[7] == 'long':
            close_time = row[6]
            balance += row[4] * (row[3] - row[2])
```

```
        elif row[7] == 'short':
            close_time = row[6]
            balance += row[4] * (row[2] - row[3])

    for row in opened_orders:
        if row[4] == 'long':
            current += row[2] * (current_price - row[1])

        elif row[4] == 'short':
            current += row[2] * (row[1] - current_price)

    insert_balance(balance + current, close_time)


def insert_balance(price, date):
    global db, selected_currency

    price = round(price, 3)
    insert_sentence = "INSERT INTO {}_BALANCE (BALANCE, DATE)
       VALUES({}, \'{}\');".format(selected_currency, price,




    execute_sentence(insert_sentence, db.cursor())


def get_liquid_value():
    global db
    global end_year
    global start_year

    mpl.style.use('classic')

    select_sentence = "SELECT DATE, BALANCE FROM {}_BALANCE;".
       format(selected_currency)

    data = pd.read_sql(select_sentence, db, parse_dates=['DATE'])

    window = int(round(data['DATE'].count() / ((end_year + 1 -
       start_year) * 20)))

    if window ==0:
```

```
        window = 1

    data['BALANCE'] = data.BALANCE.rolling(window=window).mean()

    return data


def get_total_stats():
    global db
    global selected_currency

    update_stats()

    select_sentence = """
                        SELECT SUM(OPENED_ORDERS), SUM(
                            CLOSED_ORDERS), SUM(TAKE_PROFIT_ORDERS)
                             , SUM(STOP_LOSS_ORDERS)
                         FROM {}_MONTHLY_STATS WHERE OPENED_ORDERS
                              IS NOT NULL;
                        """.format(selected_currency)
    total = execute_sentence(select_sentence, db.cursor())

    for row in total:
        opened = row[0]
        closed = row[1]
        take_profites = row[2]
        stop_losses = row[3]

    closed_percentage = closed / opened * 100

    if(closed != 0):
        tp_percentage = take_profites / closed * 100
        sl_percentage = stop_losses / closed * 100

    else:
        tp_percentage = 0.0
        sl_percentage = 0.0


    insert_sentence = "INSERT INTO {}_MONTHLY_STATS (MONTH_YEAR,
       OPENED_ORDERS, CLOSED_ORDERS, TAKE_PROFIT_ORDERS," \
                      " STOP_LOSS_ORDERS, CLOSED_ORDERS_PERCENTAGE,
                         TAKE_PROFIT_PERCENTAGE, STOP_LOSS_PERCENTAGE)
                          " \
```

```
                    "VALUES (\'EXECUTION\', \'{}\', \'{}\', \'{}\',
                       \'{}\', {}, {}, {});".format(
                       selected_currency,
```

```
    execute_sentence(insert_sentence, db.cursor())


def print_stats():
    global db

    select_sentence = "SELECT * FROM {}_MONTHLY_STATS WHERE
```

```
        CLOSED_ORDERS_PERCENTAGE IS NOT NULL " \
                        "AND {}_MONTHLY_STATS.MONTH_YEAR = \'
                            EXECUTION\';".format(selected_currency,
                            selected_currency)

    data = pd.read_sql(select_sentence, db)

    return data


def reset_database():
    global db

    delete_sentence = "DELETE FROM {}_ALL_ORDERS;".format(
        selected_currency)
    execute_sentence(delete_sentence, db.cursor())

    delete_sentence2 = "DELETE FROM {}_ORDERS;".format(
        selected_currency)
    execute_sentence(delete_sentence2, db.cursor())

    delete_sentence3 = "DELETE FROM {}_CLOSED_ORDERS;".format(
        selected_currency)
    execute_sentence(delete_sentence3, db.cursor())

    delete_sentence4 = "DELETE FROM {}_BALANCE;".format(
        selected_currency)
    execute_sentence(delete_sentence4, db.cursor())

    delete_sentence5 = "DELETE FROM {}_MONTHLY_STATS;".format(
        selected_currency)
    execute_sentence(delete_sentence5, db.cursor())


def reset_global_variables():
    global balance, opened_orders, closed_orders,
        take_profit_orders, stop_loss_orders, current_month

    balance = 0.0
    opened_orders = 0
    closed_orders = 0
    take_profit_orders = 0
    stop_loss_orders = 0
    current_month = 1
```

```
 Class historicClasses

import random
import string
import math
from math import exp
from math import log
from math import pow
from math import sqrt

from scipy.stats import norm

import DBConnection as dbc


class Agent:
    """
    Trader, manages positions
    """

    def __init__(self, original_threshold=0.01, agent_mode='long')
        :

        self.original_threshold = original_threshold

        self.agent_mode = agent_mode

        self.opened_orders_long = []
        self.opened_orders_short = []

        # unit size is the percentage of the cash that will be bid
            in each position
        self.unit_size = 10

        self.inventory = 0

        self.liquidity_indicator = LiquidityIndicator(self.
           original_threshold)

        self.events_recorder = EventsRecorder(original_threshold,
           'up')

    def trade(self, price):
```

```
        """
        this method opens new positions or closes already opened
           positions depending on intrinsics events
        directionalChangeToDown: directional change to down mode
        directionalChangeToDown: directional change to down mode
        downOvershoot: overshoot given a downwards direction
        upOvershoot: overshoot given a upwards direction
        :param price:
        :return:
        """

        self.liquidity_indicator.run(price)

        event = self.events_recorder.record_event(price)

        assert event in ('NOevent', 'upOvershoot', 'downOvershoot
           ', 'directionalChangeToUp',
                         'directionalChangeToDown'), "{} this is
                            not a valid event".format(event)
        assert self.agent_mode in ('long', 'short'), "{} not a
           valid long_short value".format(self.agent_mode)

        if event != 'NOevent':

            if self.agent_mode == 'long':

                if event == 'downOvershoot' or event == '
                   directionalChangeToUp':
                    self.open_new_order(price, event)

                elif event == 'upOvershoot':
                    self.sell_opened_positions(price, event)

            else:
                if event == 'upOvershoot' or event == '
                   directionalChangeToDown':
                    self.open_new_order(price, event)
                elif event == 'downOvershoot':
                    self.sell_opened_positions(price, event)

        return 0

    def open_new_order(self, price, event):
        if self.agent_mode == 'long':
```

```
            size_adjustment = self.liquidity_indicator.
                adjust_sizing()

            if event is 'downOvershoot':
                take_profit = self.compute_take_profit(price.ask,
                    self.events_recorder.threshold_down, 'long')
                stop_loss = self.compute_stop_loss(price.ask, self
                    .events_recorder.threshold_down, 'long')
                new_order = Order(price.ask, self.unit_size *
                    size_adjustment, price.time, self.agent_mode,
                                  self.events_recorder.
                                      threshold_up, self.
                                      events_recorder.
                                      threshold_down, event,
                                  take_profit, stop_loss, self.
                                      liquidity_indicator.liquidity
                                      , self.inventory)
            elif event is 'directionalChangeToUp':
                take_profit = self.compute_take_profit(price.ask,
                    self.events_recorder.threshold_up, 'long')
                stop_loss = self.compute_stop_loss(price.ask, self
                    .events_recorder.threshold_up, 'long')
                new_order = Order(price.ask, self.unit_size *
                    size_adjustment, price.time, self.agent_mode,
                                  self.events_recorder.
                                      threshold_up, self.
                                      events_recorder.
                                      threshold_down, event,
                                  take_profit, stop_loss, self.
                                      liquidity_indicator.liquidity
                                      , self.inventory)

            self.opened_orders_long.append(new_order)
            self.inventory += self.unit_size * size_adjustment
            self.events_recorder.adjust_thresholds(self.inventory)
            self.liquidity_indicator.adjust_thresholds(self.
                inventory)
            dbc.create_database_buy_order(new_order)

        else:
            size_adjustment = self.liquidity_indicator.
                adjust_sizing()

            if event is 'upOvershoot':
```

```
                take_profit = self.compute_take_profit(price.ask,
                    self.events_recorder.threshold_up, 'short')
                stop_loss = self.compute_stop_loss(price.ask, self
                    .events_recorder.threshold_up, 'short')
                new_order = Order(price.ask, self.unit_size *
                    size_adjustment, price.time, self.agent_mode,
                                  self.events_recorder.
                                      threshold_up, self.
                                      events_recorder.
                                      threshold_down, event,
                                  take_profit, stop_loss, self.
                                      liquidity_indicator.liquidity
                                      , self.inventory)
            elif event is 'directionalChangeToDown':
                take_profit = self.compute_take_profit(price.ask,
                    self.events_recorder.threshold_down, 'short')
                stop_loss = self.compute_stop_loss(price.ask, self
                    .events_recorder.threshold_down, 'short')
                new_order = Order(price.ask, self.unit_size *
                    size_adjustment, price.time, self.agent_mode,
                                  self.events_recorder.
                                      threshold_up, self.
                                      events_recorder.
                                      threshold_down, event,
                                  take_profit, stop_loss, self.
                                      liquidity_indicator.liquidity
                                      , self.inventory)
            self.opened_orders_short.append(new_order)
            self.inventory -= self.unit_size * size_adjustment
            self.events_recorder.adjust_thresholds(self.inventory)
            self.liquidity_indicator.adjust_thresholds(self.
                inventory)
            dbc.create_database_buy_order(new_order)

    def sell_opened_positions(self, price, event):
        if self.agent_mode == 'long':
            for order in self.opened_orders_long:
                if price.bid >= order.take_profit:
                    self.opened_orders_long.remove(order)
                    self.inventory -= order.volume
                    self.events_recorder.adjust_thresholds(self.
                        inventory)
                    self.liquidity_indicator.adjust_thresholds(
                        self.inventory)
```

```
                    dbc.create_database_sell_order(order.order_id,
                        price.bid, price.time, event, 'TakeProfit
                    ')

                elif price.bid <= order.stop_loss:
                    self.opened_orders_long.remove(order)
                    self.inventory -= order.volume
                    self.events_recorder.adjust_thresholds(self.
                        inventory)
                    self.liquidity_indicator.adjust_thresholds(
                        self.inventory)
                    dbc.create_database_sell_order(order.order_id,
                        price.bid, price.time, event, 'StopLoss')
        else:
            for order in self.opened_orders_short:
                if price.ask <= order.take_profit:
                    self.opened_orders_short.remove(order)
                    self.inventory += order.volume
                    self.events_recorder.adjust_thresholds(self.
                        inventory)
                    self.liquidity_indicator.adjust_thresholds(
                        self.inventory)
                    dbc.create_database_sell_order(order.order_id,
                        price.ask, price.time, event, 'TakeProfit
                    ')

                elif price.ask >= order.stop_loss:
                    self.opened_orders_short.remove(order)
                    self.inventory += order.volume
                    self.events_recorder.adjust_thresholds(self.
                        inventory)
                    self.liquidity_indicator.adjust_thresholds(
                        self.inventory)
                    dbc.create_database_sell_order(order.order_id,
                        price.ask, price.time, event, 'StopLoss')

    def compute_take_profit(self, price, threshold, mode):
            if mode is 'long':
                return exp(log(price) + threshold)
            elif mode is 'short':
                return exp(log(price) - threshold)

    def compute_stop_loss(self, price, threshold, mode):
        if mode is 'long':
```

```
            return price * (1 - (0.15 - (0.1 - (self.
                liquidity_indicator.liquidity * threshold * 10))))
        elif mode is 'short':
            return price * (1 + (0.15 - (0.1 - (self.
                liquidity_indicator.liquidity * threshold * 10))))


class EventsRecorder:
    """
    Records events (overshoots and directional changes)
    """

    def __init__(self, original_threshold, market_mode='up'):

        self.original_threshold = original_threshold
        self.threshold_up = original_threshold
        self.threshold_down = original_threshold

        self.market_mode = market_mode
        self.reference = None
        self.extreme = None
        self.expected_overshoot_price = None
        self.expected_directional_change_price = None
        self.initialized = False

    def record_event(self, price):
        """
        Records an event given a price
        :param price
        :return: NOevent, directionalChangeToDown,
            directionalChangeToDown, downOvershoot, upOvershoot
        """

        assert self.market_mode in ('up', 'down'), '{} is not a
            valid market mode'.format(self.market_mode)
        if not self.initialized:
            self.initialized = True
            self.reference = self.extreme = price.get_mid()
            self.compute_expected_directional_change()
            self.compute_expected_overshoot()
            return 'NOevent'

        if self.market_mode == 'up':
```

```
            if price.get_bid() > self.extreme:
                self.extreme = price.get_bid()
                self.compute_expected_directional_change()
                if price.get_bid() > self.expected_overshoot_price
                    :
                    self.reference = self.extreme
                    self.compute_expected_overshoot()
                    return 'upOvershoot'

            elif price.get_ask() <= self.
                expected_directional_change_price:
                self.reference = self.extreme = price.get_ask()
                self.market_mode = 'down'
                self.compute_expected_directional_change()
                self.compute_expected_overshoot()
                return 'directionalChangeToDown'

        else:
            if price.get_ask() < self.extreme:
                self.extreme = price.get_ask()
                self.compute_expected_directional_change()

                if price.get_ask() < self.expected_overshoot_price
                    :
                    self.reference = self.extreme
                    self.compute_expected_overshoot()
                    return 'downOvershoot'

            elif price.get_bid() >= self.
                expected_directional_change_price:
                self.reference = self.extreme = price.get_bid()
                self.market_mode = 'up'
                self.compute_expected_directional_change()
                self.compute_expected_overshoot()
                return 'directionalChangeToUp'

        return 'NOevent'

    def compute_expected_overshoot(self):
        assert self.market_mode in ('up', 'down'), '{} not a valid
            market mode in method get_expected_OS'.format(self.
            market_mode)
        if self.market_mode == 'up':
            self.expected_overshoot_price = exp(log(self.reference
```

```
                ) + self.threshold_up)
        else:
            self.expected_overshoot_price = exp(log(self.reference
                ) - self.threshold_down)

    def compute_expected_directional_change(self):
        assert self.market_mode in ('up', 'down'), '{} not a valid
            market mode in method get_expected_DC '.format(self.
           market_mode)
        if self.market_mode == 'up':
            self.expected_directional_change_price = exp(log(self.
                reference) - self.threshold_down)
        else:
            self.expected_directional_change_price = exp(log(self.
                reference) + self.threshold_up)

    def adjust_thresholds(self, inventory):

        #LONG
        if inventory >= 150 and inventory < 300:
            self.threshold_up = self.original_threshold * 0.75
            self.threshold_down = self.original_threshold * 1.5
        elif inventory >= 300:
            self.threshold_up = self.original_threshold * 0.5
            self.threshold_down = self.original_threshold * 2

        #SHORT
        elif inventory > -300 and inventory <= -150:
            self.threshold_up = self.original_threshold * 1.5
            self.threshold_down = self.original_threshold * 0.75
        elif inventory <= -300:
            self.threshold_up = self.original_threshold * 2
            self.threshold_down = self.original_threshold * 0.5

        else:
            self.threshold_up = self.original_threshold
            self.threshold_down = self.original_threshold


class Price:
    """
    This class represents the price object, which will be passed
       to the agent with every new
    tick of the market.
```

```
    """

    def __init__(self, id, ask, bid, time):
        self.id = id
        self.ask = ask
        self.bid = bid
        self.time = time

    def clone(self):
        return Price(self.id, self.ask, self.bid, self.time)

    def get_id(self):
        return self.id

    def get_ask(self):
        return self.ask

    def get_bid(self):
        return self.bid

    def get_spread(self):
        return self.ask - self.bid

    def get_mid(self):
        return (self.ask + self.bid) / 2

    def get_time(self):
        return self.time


class LiquidityIndicator:

    def __init__(self, original_threshold):
        self.liquidity = 0.0
        self.original_threshold = original_threshold
        self.threshold_up = original_threshold
        self.threshold_down = original_threshold
        self.K = 50.0
        self.alpha_weight = math.exp(-2.0/(self.K + 1.0))
        self.H1 = self.compute_h1()
        self.H2 = self.compute_h2()
        self.surprise = 0.0
        self.initialized = False
        self.extreme = None
```

```
        self.reference = None
        self.mode = 'up'

    def compute_h1(self):

        return -exp(-self.original_threshold * 2.52579 / self.
            original_threshold) * log(exp(-self.original_threshold
            * 2.52579 / self.original_threshold)) - (1.0 - exp(-
            self.original_threshold * 2.52579 / self.
            original_threshold)) * log(1 - exp(-self.
            original_threshold * 2.52579 / self.original_threshold)
            )

    def compute_h2(self):

        return exp(-self.original_threshold * 2.52579 / self.
            original_threshold) * pow(log(exp(-self.
            original_threshold * 2.52579 / self.original_threshold)
            ), 2) - (1.0 - exp(-self.original_threshold * 2.52579 /
             self.original_threshold)) * pow(log(1 - exp(-self.
            original_threshold * 2.52579 / self.original_threshold)
            ),2) - self.H1 * self.H1

    def adjust_sizing(self):

        assert 0 <= self.liquidity <= 1, "{} not a valid value".
            format(self.liquidity)

        if 0.1 < self.liquidity < 0.5:
            return 0.5
        elif self.liquidity <= 0.1:
            return 0.1
        else:
            return 1

    def compute_surprise(self, event):
        if event == 'directionalChange':
            return self.alpha_weight * 0.08338161 + (1.0 - self.
                alpha_weight) * self.surprise
        elif event == 'overshoot':
            return self.alpha_weight * 2.52579 + (1.0 - self.
                alpha_weight) * self.surprise

    def compute_liquidity(self, event):
```

```
        if event != 'NOevent':

            self.surprise = self.compute_surprise(event)

            self.liquidity = 1.0 - norm.cdf(sqrt(self.K) * (self.
               surprise - self.H1) / sqrt(self.H2))


    def run(self, price):

        if not self.initialized:
            self.extreme = self.reference = price.get_mid()
            self.initialized = True
            self.compute_liquidity('NOevent')

        if self.mode is 'down':
            if math.log(price.get_bid() / self.extreme) >= self.
               threshold_up:
                self.mode = 'up'
                self.extreme = price.get_bid()
                self.reference = price.get_bid()
                self.compute_liquidity('directionalChange')

            if price.get_ask() < self.extreme:
                self.extreme = price.get_ask()

            if math.log(self.reference/self.extreme) >= self.
               threshold_down * 2.52579:
                self.reference = self.extreme
                self.compute_liquidity('overshoot')

        elif self.mode is 'up':
            if math.log(price.get_ask()/self.extreme) <= -self.
               threshold_down:
                self.mode = 'down'
                self.extreme = price.get_ask()
                self.reference = price.get_ask()
                self.compute_liquidity('directionalChange')

            if price.get_bid() > self.extreme:
                self.extreme = price.get_bid()

            if math.log(self.reference / self.extreme) <= -self.
```

```
                threshold_up * 2.52579:
                 self.reference = self.extreme
                 self.compute_liquidity('overshoot')


    def adjust_thresholds(self, inventory):
        #LONG
        if inventory >= 150 and inventory < 300:
            self.threshold_up = self.original_threshold * 0.75
            self.threshold_down = self.original_threshold * 1.5
        elif inventory >= 300:
            self.threshold_up = self.original_threshold * 0.5
            self.threshold_down = self.original_threshold * 2

        #SHORT
        elif inventory > -300 and inventory <= -150:
            self.threshold_up = self.original_threshold * 1.5
            self.threshold_down = self.original_threshold * 0.75
        elif inventory <= -300:
            self.threshold_up = self.original_threshold * 2
            self.threshold_down = self.original_threshold * 0.5

        else:
            self.threshold_up = self.original_threshold
            self.threshold_down = self.original_threshold


class Order:
    """
    This class represents a position
    Positions will be opened and closed by agents when operating
    """

    def __init__(self, open_price=0.0, volume=0, time=0.0,
       agent_mode='long', threshold_up=0.01, threshold_down=0.01,
       event_of_creation='UpOvershoot', take_profit=0.0, stop_loss
       =0.0, liquidity = 0.0, inventory = 0):
        self.order_id = ''.join([random.choice(string.
           ascii_letters + string.digits) for _ in range(25)])
        self.open_price = open_price
        self.volume = volume
        self.time = time
        self.agent_mode = agent_mode
        self.threshold_up = threshold_up
```

```
        self.threshold_down = threshold_down
        self.event_of_creation = event_of_creation
        self.take_profit = take_profit
        self.stop_loss = stop_loss
        self.liquidity = liquidity
        self.inventory = inventory

\end{verbatim}
```

Class classesBroker

```
from math import exp
from math import log
from math import pow
from math import sqrt
import math


class Agent:
    """
    Trader, manages positions
    """

    def __init__(self, con, currency_pair='EUR/USD',
       original_threshold=0.01, agent_mode='long'):
        self.original_threshold = original_threshold
        self.agent_mode = agent_mode

        self.currency_pair = currency_pair

        # unit size is the percentage of the cash that will be bid
            in each position
        self.unit_size = 100

        self.inventory = 0

        self.liquidity_indicator = LiquidityIndicator(self.
           original_threshold)

        self.events_recorder = EventsRecorder(self.
           original_threshold)

        self.broker = con
```

```
def trade(self, last_price):
    """
    this method opens new positions or closes already opened
       positions depending on intrinsics events
    directionalChangeToDown: directional change to down mode
    directionalChangeToDown: directional change to down mode
    downOvershoot: overshoot given a downwards direction
    upOvershoot: overshoot given a upwards direction
    :return:
    """

    print(f'connection status: {self.broker.is_connected()}')

    print(f'last price is : {last_price}')
    # call liquidity here
    self.liquidity_indicator.run(last_price)

    event = self.events_recorder.record_event(last_price)
    print(f'event: {event}')

    assert event in ('NOevent', 'upOvershoot', 'downOvershoot
       ', 'directionalChangeToUp',
                     'directionalChangeToDown'), "{} this is
                        not a valid event".format(event)
    assert self.agent_mode in ('long', 'short'), "{} not a
       valid long_short value".format(self.agent_mode)

    if event != 'NOevent':

        if self.agent_mode == 'long':

            if event == 'downOvershoot' or event == '
               directionalChangeToUp':
                self.open_new_order()
            elif event == 'upOvershoot':
                self.sell_opened_positions(last_price)
            else:
                pass

        else:
            if event == 'upOvershoot' or event == '
               directionalChangeToDown':
                self.open_new_order()
```

```
                elif event == 'downOvershoot':
                    self.sell_opened_positions(last_price)
                else:
                    pass

        return 0

    def open_new_order(self):

        if self.agent_mode == 'long':
            size = self.liquidity_indicator.adjust_sizing()
            self.broker.create_market_buy_order(self.currency_pair
                , self.unit_size * size)
            self.inventory += self.unit_size * size
        else:
            size = self.liquidity_indicator.adjust_sizing()
            self.broker.create_market_sell_order(self.
                currency_pair, self.unit_size * size)
            self.inventory -= self.unit_size * size

        self.events_recorder.adjust_thresholds(self.inventory)
        self.liquidity_indicator.adjust_thresholds(self.inventory)


    def sell_opened_positions(self, last_price):
        orders_list = self.broker.get_open_positions(kind='list')
        for order in orders_list:
            print(f'inside for loop in sell order with pl: {order
                ["visiblePL"]}')
            if self.stop_loss(order, last_price) or self.profit(
                order):
                self.broker.close_trade(order['tradeId'], order['
                    amountK'])
                self.adjust_inventory(order)

    def stop_loss(self, order, last_price):
        if order['isBuy']:
            stop = order['open'] * (1 - (0.15 - (0.1 - (self.
                liquidity_indicator.liquidity * self.
                events_recorder.threshold_up * 10))))
            return True if last_price['Ask'] < stop else False
        else:
            stop = order['open'] * (1 + (0.15 - (0.1 - (self.
                liquidity_indicator.liquidity * self.
```

```
                events_recorder.threshold_down * 10))))
            return True if last_price['Ask'] > stop else False

    def profit(self, order):
        return True if order['visiblePL'] > 0.5 else False

    def adjust_inventory(self, order):
        if order['isBuy']:
            self.inventory += order['amountK']
        else:
            self.inventory -= order['amountK']


class EventsRecorder:
    """
    Records events (overshoots and directional changes)
    """

    def __init__(self, original_threshold):
        self.original_threshold = original_threshold
        self.threshold_up = original_threshold
        self.threshold_down = original_threshold
        self.market_mode = 'up'
        self.reference = None
        self.extreme = None
        self.expected_overshoot_price = None
        self.expected_directional_change_price = None
        self.initialized = False

    def record_event(self, last_price):
        """
        Records an event given a price
        :param last_price
        :return: NOevent, directionalChangeToDown,
           directionalChangeToDown, downOvershoot, upOvershoot
        """

        assert self.market_mode in ('up', 'down'), '{} is not a
           valid market mode'.format(self.market_mode)
        if not self.initialized:
            self.initialized = True
            self.reference = self.extreme = (last_price.Bid +
               last_price.Ask) / 2
            self.compute_expected_directional_change()
```

```
            self.compute_expected_overshoot()
            return 'NOevent'

        if self.market_mode == 'up':

            if last_price.Bid > self.extreme:
                self.extreme = last_price.Bid
                self.compute_expected_directional_change()

                if last_price.Bid > self.expected_overshoot_price:
                    self.reference = self.extreme
                    self.compute_expected_overshoot()
                    return 'upOvershoot'

            elif last_price.Ask <= self.
               expected_directional_change_price:
                self.reference = self.extreme = last_price.Ask
                self.market_mode = 'down'
                self.compute_expected_directional_change()
                self.compute_expected_overshoot()
                return 'directionalChangeToDown'
        else:
            if last_price.Ask < self.extreme:
                self.extreme = last_price.Ask
                self.compute_expected_directional_change()

                if last_price.Ask < self.expected_overshoot_price:
                    self.reference = self.extreme
                    self.compute_expected_overshoot()
                    return 'downOvershoot'

            elif last_price.Bid >= self.
               expected_directional_change_price:
                self.reference = self.extreme = last_price.Bid
                self.market_mode = 'up'
                self.compute_expected_directional_change()
                self.compute_expected_overshoot()
                return 'directionalChangeToUp'

        return 'NOevent'

    def compute_expected_overshoot(self):
        assert self.market_mode in ('up', 'down'), f'{self.
           market_mode} not a valid market mode in method
```

```
            get_expected_OS'
        if self.market_mode == 'up':
            self.expected_overshoot_price = exp(log(self.reference
                ) + self.threshold_up)
        else:
            self.expected_overshoot_price = exp(log(self.reference
                ) - self.threshold_down)

    def compute_expected_directional_change(self):
        assert self.market_mode in ('up', 'down'), f'{self.
            market_mode} not a valid market mode in method
            get_expected_DC'
        if self.market_mode == 'up':
            self.expected_directional_change_price = exp(log(self.
                reference) - self.threshold_down)
        else:
            self.expected_directional_change_price = exp(log(self.
                reference) + self.threshold_up)

    def adjust_thresholds(self, inventory):
        if inventory >= 1500 and inventory < 3000:
            self.threshold_up = self.original_threshold * 0.75
            self.threshold_down = self.original_threshold * 1.5
        elif inventory >= 3000:
            self.threshold_up = self.original_threshold * 0.5
            self.threshold_down = self.original_threshold * 2

        elif inventory > -3000 and inventory <= -1500:
            self.threshold_up = self.original_threshold * 1.5
            self.threshold_down = self.original_threshold * 0.75
        elif inventory <= -3000:
            self.threshold_up = self.original_threshold * 2
            self.threshold_down = self.original_threshold * 0.5
        else:
            self.threshold_up = self.original_threshold
            self.threshold_down = self.original_threshold


class LiquidityIndicator:

    def __init__(self, original_threshold):
        self.original_threshold = original_threshold
        self.threshold_up = original_threshold
        self.threshold_down = original_threshold
```

```
        self.liquidity = 0.0
        self.K = 50.0
        self.alpha_weight = math.exp(-2.0 / (self.K + 1.0))
        self.H1 = self.compute_h1()
        self.H2 = self.compute_h2()
        self.surprise = 0.0
        self.initialized = False
        self.extreme = None
        self.reference = None
        self.mode = 'up'

    def compute_h1(self):
        return -exp(-self.original_threshold * 2.52579 / self.
            original_threshold) * log(
             exp(-self.original_threshold * 2.52579 / self.
                original_threshold)) - (
                        1.0 - exp(-self.original_threshold *
                            2.52579 / self.original_threshold)) *
                            log(
             1 - exp(-self.original_threshold * 2.52579 / self.
                original_threshold))

    def compute_h2(self):

        return exp(-self.original_threshold * 2.52579 / self.
            original_threshold) * pow(
             log(exp(-self.original_threshold * 2.52579 / self.
                original_threshold)), 2.0) - (
                        1.0 - exp(-self.original_threshold *
                            2.52579 / self.original_threshold)) *
                            pow(
             log(1.0 - exp(-self.original_threshold * 2.52579 /
                self.original_threshold)), 2.0) - self.H1 * self.H1

    def adjust_sizing(self):

        assert 0 <= self.liquidity <= 1, f'{self.liquidity} not a
            valid value'

        if 0.1 < self.liquidity < 0.5:
            return 0.5
        elif self.liquidity <= 0.1:
            return 0.1
        else:
```

```
        return 1

    def compute_surprise(self, event):
        if event == 'directionalChange':
            return self.alpha_weight * 0.08338161 + (1.0 - self.
                alpha_weight) * self.surprise
        else:
            return self.alpha_weight * 2.52579 + (1.0 - self.
                alpha_weight) * self.surprise

    def compute_liquidity(self, event):

        if event != 'NOevent':
            self.surprise = self.compute_surprise(event)

            self.liquidity = 1.0 - self.
                normal_distribution_cumulative(
                 sqrt(self.K) * (self.surprise - self.H1) / sqrt(
                     self.H2))

    def normal_distribution_cumulative(self, x):

        if x > 6.0:
            return 1.0
        if x < -6.0:
            return 0.0

        b1 = 0.31938153
        b2 = -0.356563782
        b3 = 1.781477937
        b4 = -1.821255978
        b5 = 1.330274429
        p = 0.2316419
        c2 = 0.3989423

        a = abs(x)
        t = 1.0 / (1.0 + a * p)
        b = c2 * exp((-x) * (x / 2.0))
        n = ((((b5 * t + b4) * t + b3) * t + b2) * t + b1) * t
        n = 1.0 - b * n

        if x < 0.0:
            n = 1.0 - n
```

```
        return n

    def run(self, last_price):

        if not self.initialized:
            self.extreme = self.reference = (last_price.Ask +
               last_price.Bid) / 2
            self.initialized = True
            self.compute_liquidity('NOevent')

        if self.mode is 'down':
            if math.log(last_price.Bid / self.extreme) >= self.
               threshold_up:
                self.mode = 'up'
                self.extreme = last_price.Bid
                self.reference = last_price.Bid
                self.compute_liquidity('directionalChange')

            if last_price.Ask < self.extreme:
                self.extreme = last_price.Ask

            if math.log(self.reference / self.extreme) >= self.
               threshold_down * 2.52579:
                self.reference = self.extreme
                self.compute_liquidity('overshoot')

        elif self.mode is 'up':
            if math.log(last_price.Ask / self.extreme) <= -self.
               threshold_down:
                self.mode = 'down'
                self.extreme = last_price.Ask
                self.reference = last_price.Ask
                self.compute_liquidity('directionalChange')

            if last_price.Bid > self.extreme:
                self.extreme = last_price.Bid

            if math.log(self.reference / self.extreme) <= -self.
               threshold_up * 2.52579:
                self.reference = self.extreme
                self.compute_liquidity('overshoot')

    def adjust_thresholds(self, inventory):
```

```
if inventory >= 1500 and inventory < 3000:
    self.threshold_up = self.original_threshold * 0.75
    self.threshold_down = self.original_threshold * 1.5
elif inventory >= 3000:
    self.threshold_up = self.original_threshold * 0.5
    self.threshold_down = self.original_threshold * 2

elif inventory > -3000 and inventory <= -1500:
    self.threshold_up = self.original_threshold * 1.5
    self.threshold_down = self.original_threshold * 0.75
elif inventory <= -3000:
    self.threshold_up = self.original_threshold * 2
    self.threshold_down = self.original_threshold * 0.5

else:
    self.threshold_up = self.original_threshold
    self.threshold_down = self.original_threshold
```

Results of the Efficiency Tests

AUD/USD

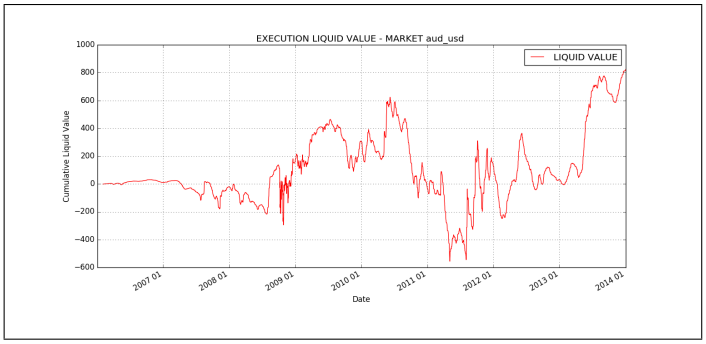


Figure 1: Results of the Version 1 - AUD/USD

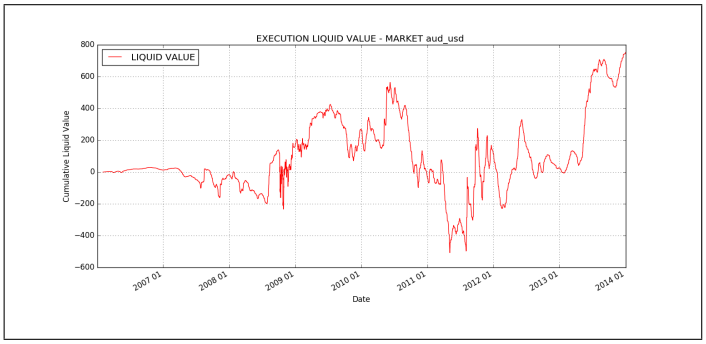


Figure 2: Results of the Version 2 - AUD/USD

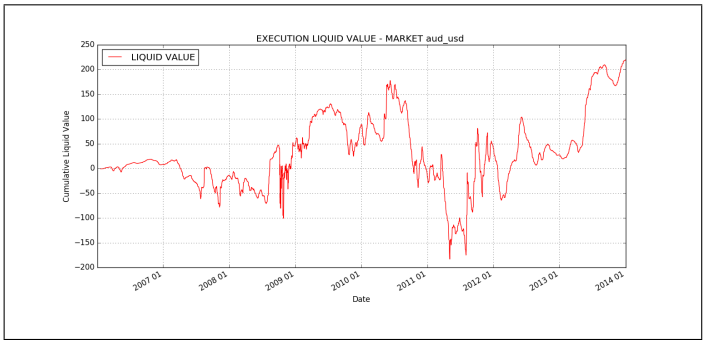


Figure 3: Results of the Version 3 - AUD/USD

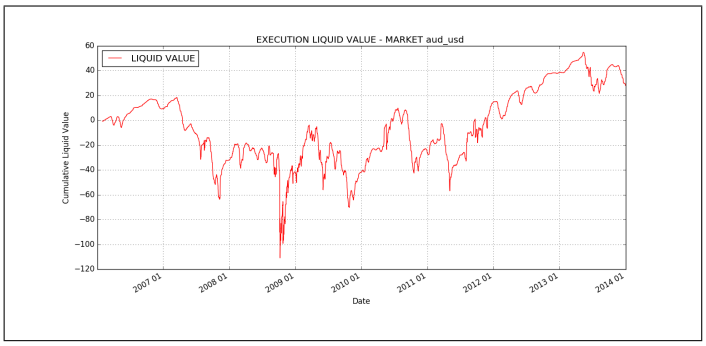


Figure 4: Results of the Version 4 - AUD/USD

EUR/CHF

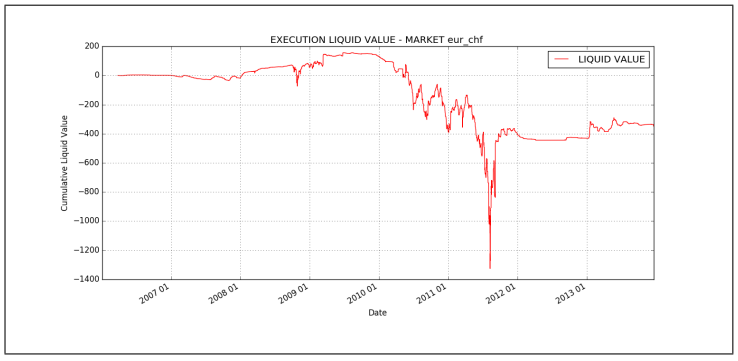


Figure 5: Results of the Version 1 - EUR/CHF

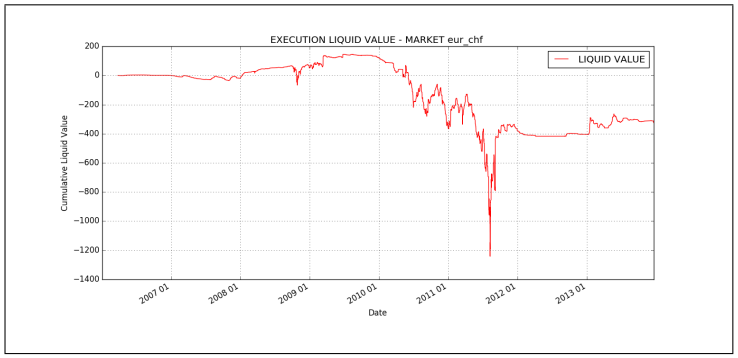


Figure 6: Results of the Version 2 - EUR/CHF

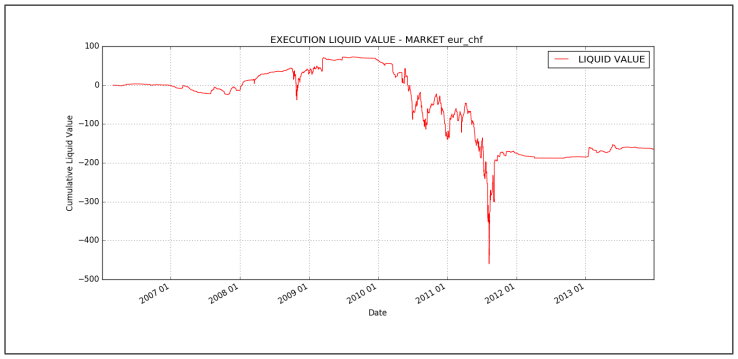


Figure 7: Results of the Version 3 - EUR/CHF

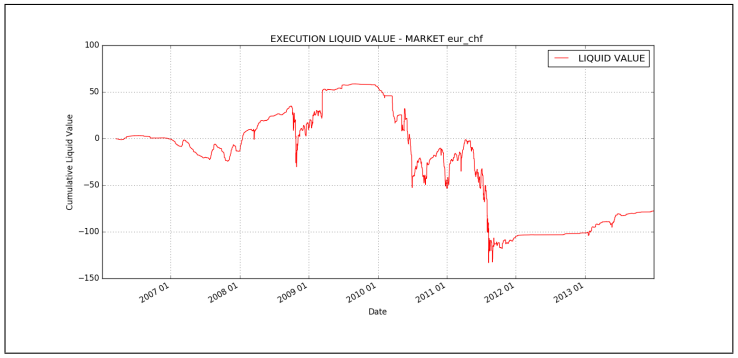


Figure 8: Results of the Version 4 - EUR/CHF

EUR/GBP

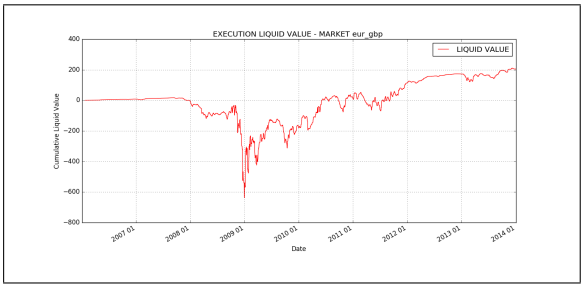


Figure 9: Results of the Version 1 - EUR/GBP

Figure 10: Results of the Version 2 - EUR/GBP

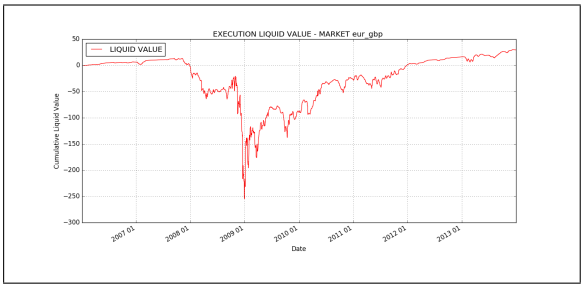


Figure 11: Results of the Version 3 - EUR/GBP

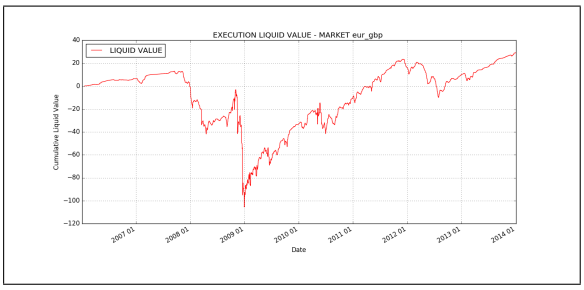


Figure 12: Results of the Version 4 - EUR/GBP

EUR/JPY

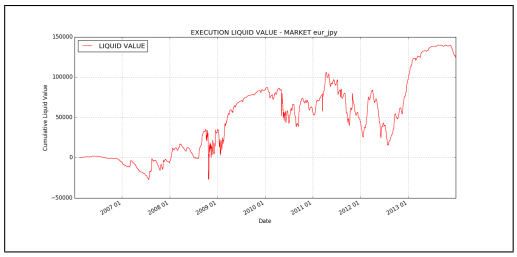


Figure 13: Results of the Version 1 - EUR/JPY

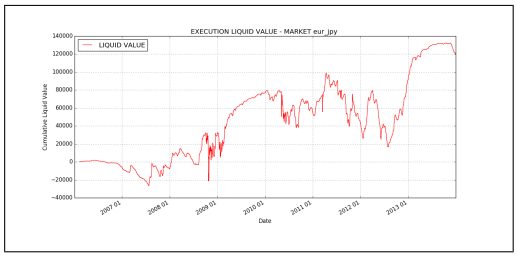


Figure 14: Results of the Version 2 - EUR/JPY

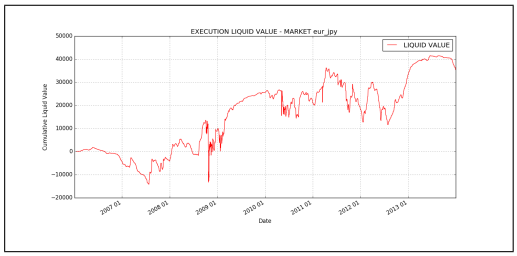


Figure 15: Results of the Version 3 - EUR/JPY

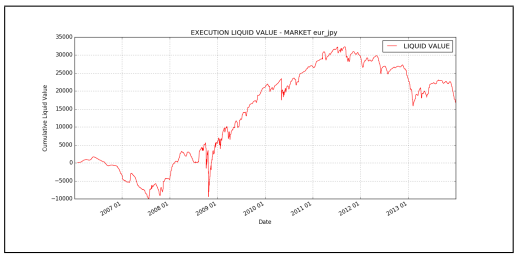


Figure 16: Results of the Version 4 - EUR/JPY

EUR/USD

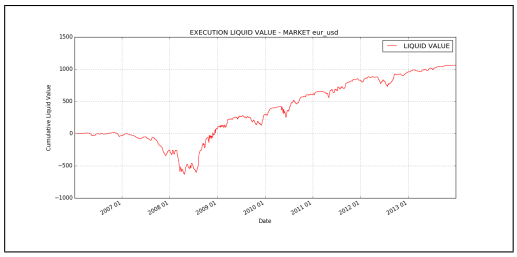


Figure 17: Results of the Version 1 - EUR/USD

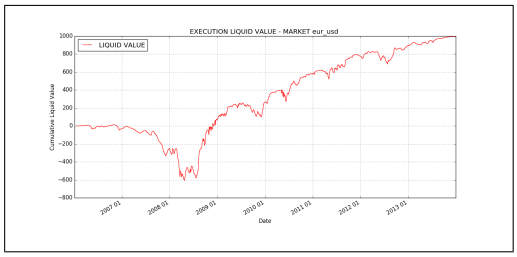


Figure 18: Results of the Version 2 - EUR/USD

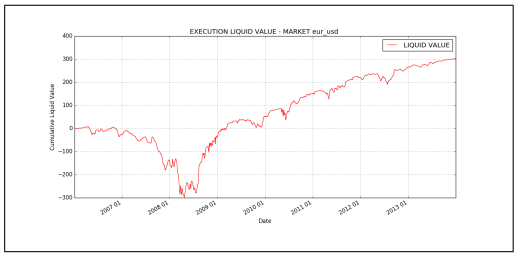


Figure 19: Results of the Version 3 - EUR/USD

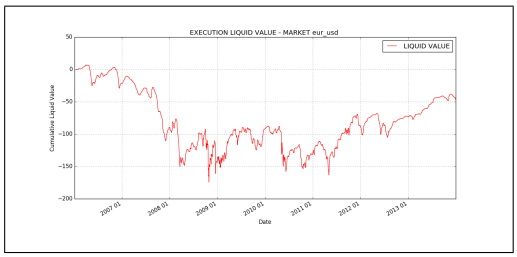


Figure 20: Results of the Version 4 - EUR/USD

GBP/USD

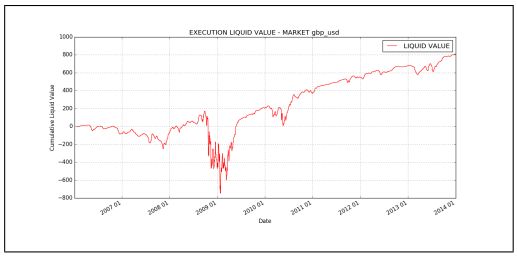


Figure 21: Results of the Version 1 - GBP/USD

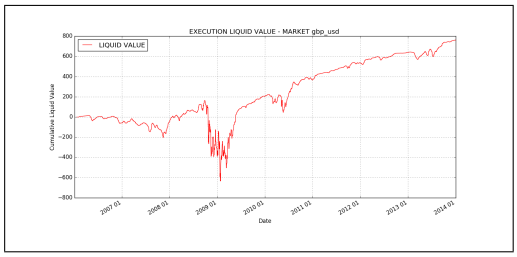


Figure 22: Results of the Version 2 - GBP/USD

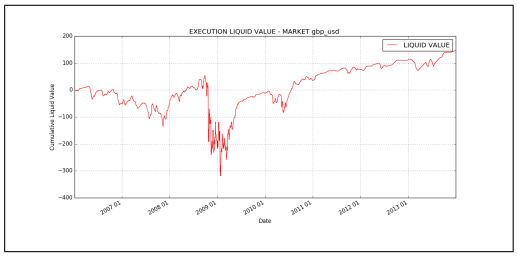


Figure 23: Results of the Version 3 - GBP/USD

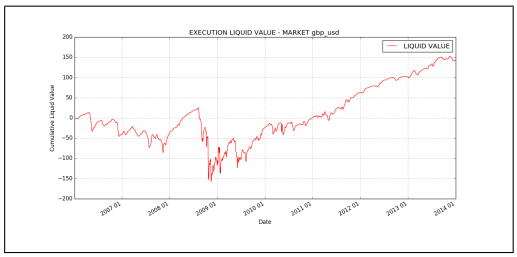


Figure 24: Results of the Version 4 - GBP/USD

USD/CAD

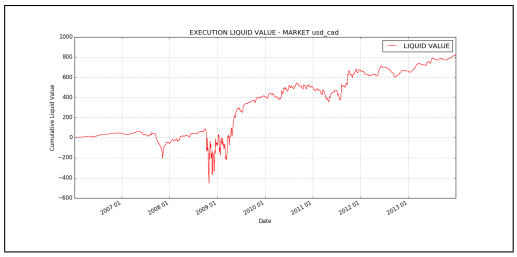


Figure 25: Results of the Version 1 - USD/CAD

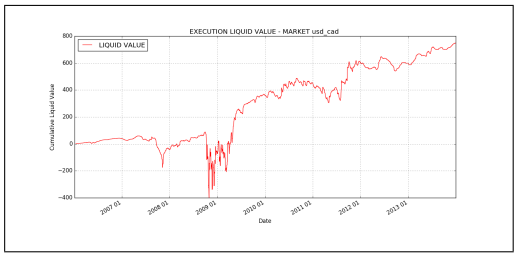


Figure 26: Results of the Version 2 - USD/CAD

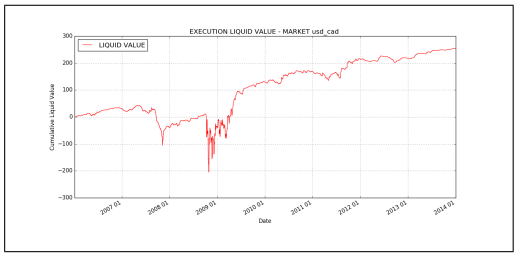


Figure 27: Results of the Version 3 - USD/CAD

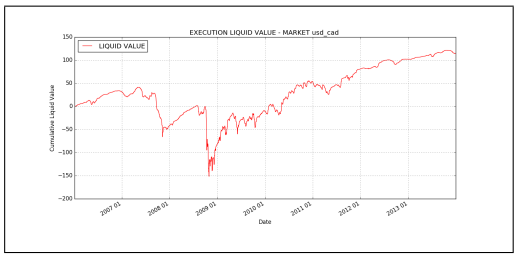


Figure 28: Results of the Version 4 - USD/CAD

USD/JPY

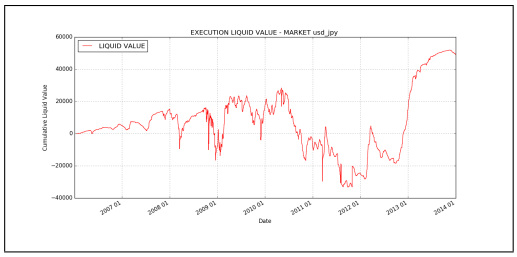


Figure 29: Results of the Version 1 - USD/JPY

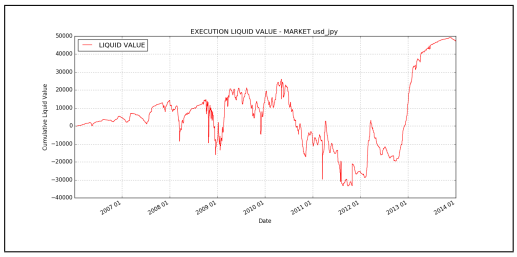


Figure 30: Results of the Version 2 - USD/JPY

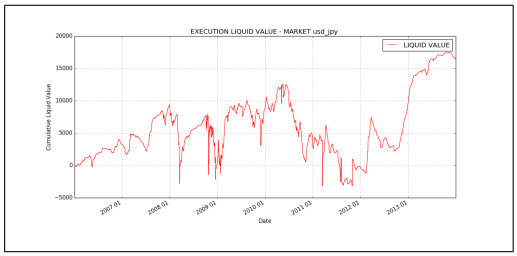


Figure 31: Results of the Version 3 - USD/JPY

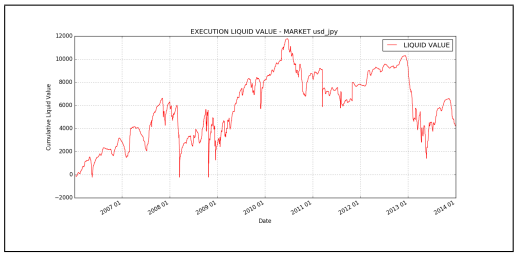


Figure 32: Results of the Version 4 - USD/JPY

CHF/JPY

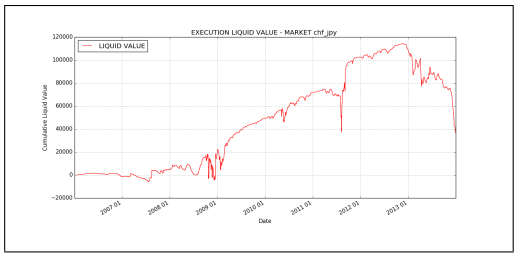


Figure 33: Results of the Version 1 - CHF/JPY

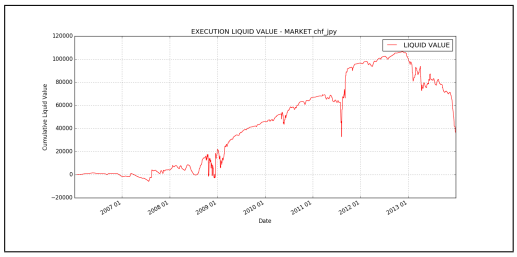


Figure 34: Results of the Version 2 - CHF/JPY

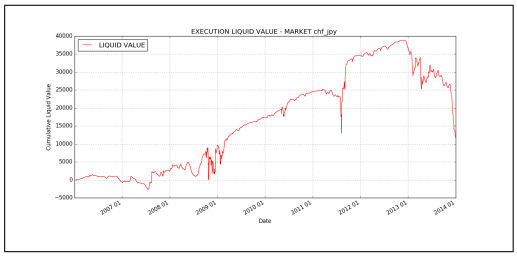


Figure 35: Results of the Version 3 - CHF/JPY

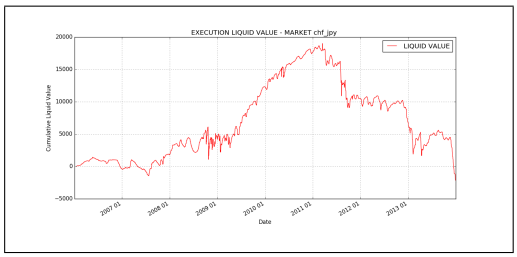


Figure 36: Results of the Version 4 - CHF/JPY

EUR/CAD

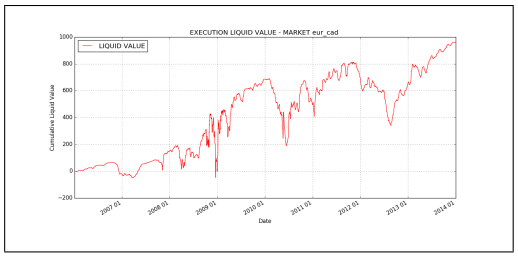


Figure 37: Results of the Version 1 - EUR/CAD

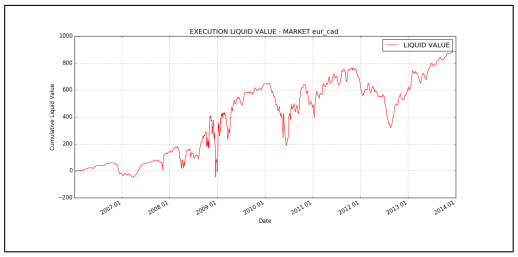


Figure 38: Results of the Version 2 - EUR/CAD

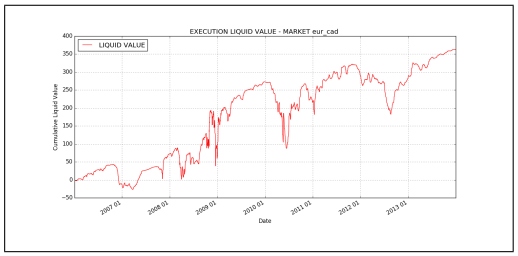


Figure 39: Results of the Version 3 - EUR/CAD

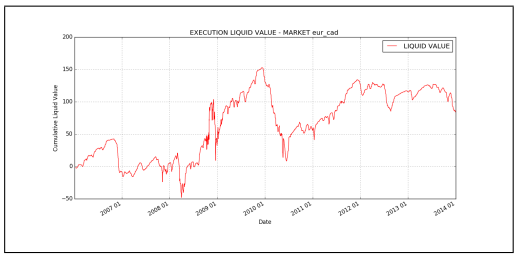


Figure 40: Results of the Version 4 - EUR/CAD

GBP/CHF

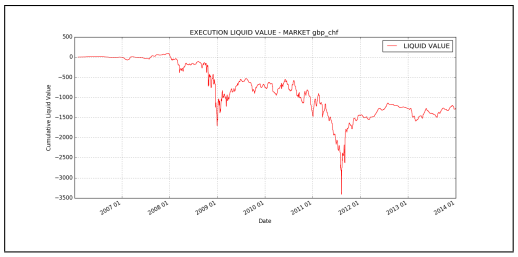


Figure 41: Results of the Version 1 - GBP/CHF

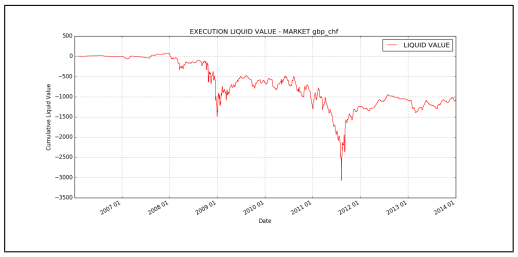


Figure 42: Results of the Version 2 - GBP/CHF

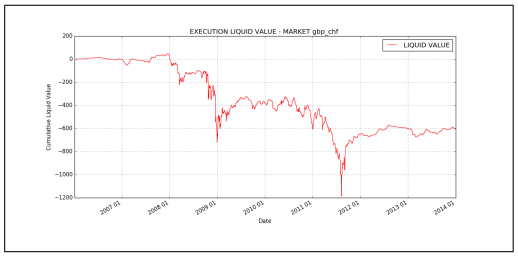


Figure 43: Results of the Version 3 - GBP/CHF

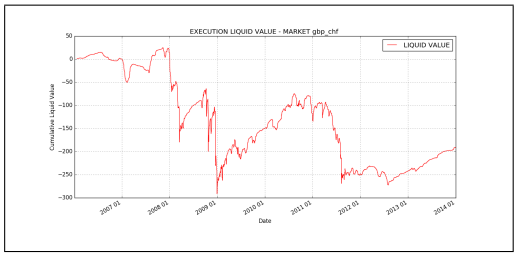


Figure 44: Results of the Version 4 - GBP/CHF

GBP/JPY

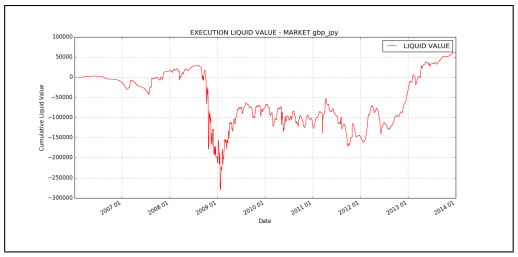


Figure 45: Results of the Version 1 - GBP/JPY

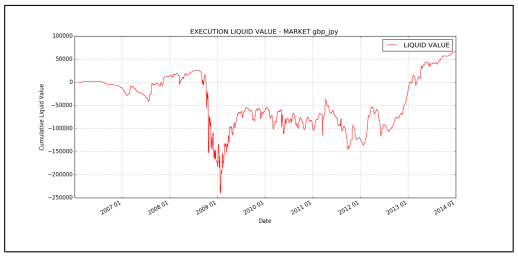


Figure 46: Results of the Version 2 - GBP/JPY

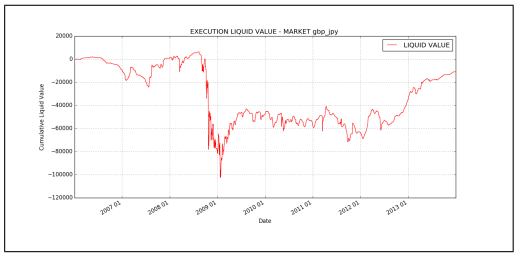


Figure 47: Results of the Version 3 - GBP/JPY

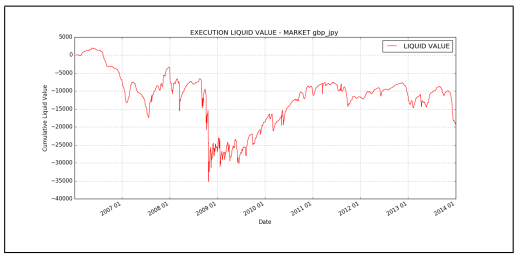


Figure 48: Results of the Version 4 - GBP/JPY

NZD/JPY

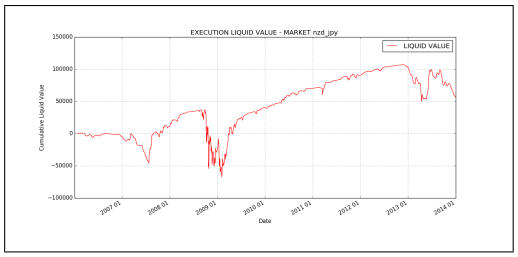


Figure 49: Results of the Version 1 - NZD/JPY

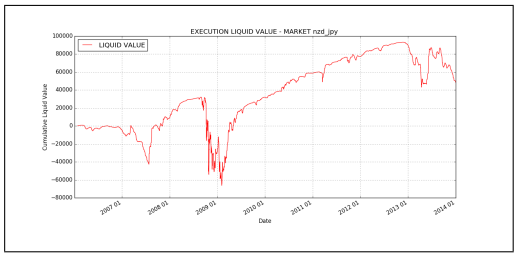


Figure 50: Results of the Version 2 - NZD/JPY

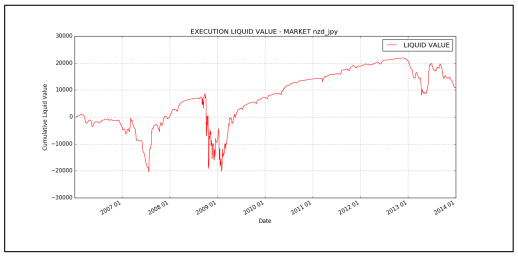


Figure 51: Results of the Version 3 - NZD/JPY

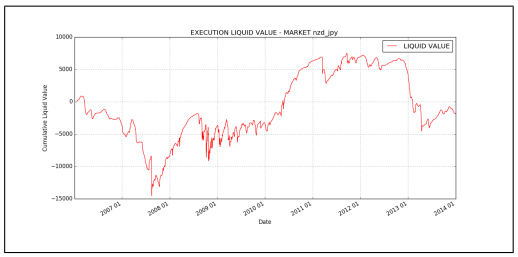


Figure 52: Results of the Version 4 - NZD/JPY

NZD/USD

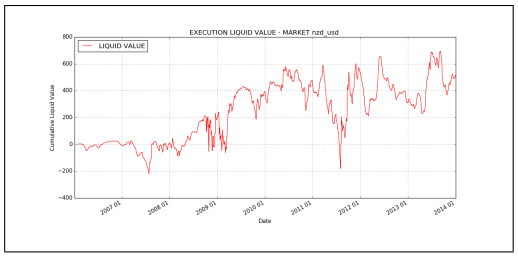


Figure 53: Results of the Version 1 - NZD/USD

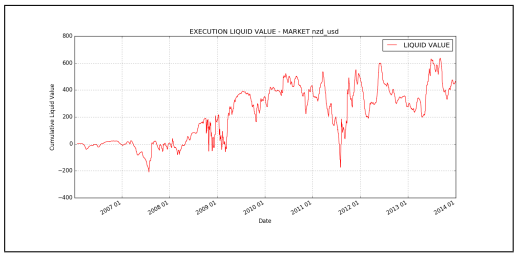


Figure 54: Results of the Version 2 - NZD/USD

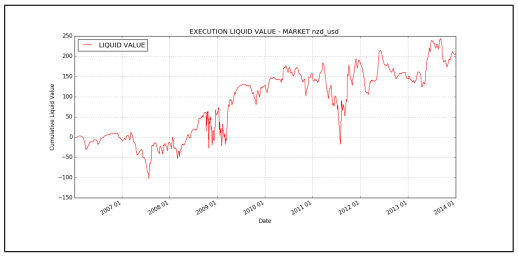


Figure 55: Results of the Version 3 - NZD/USD

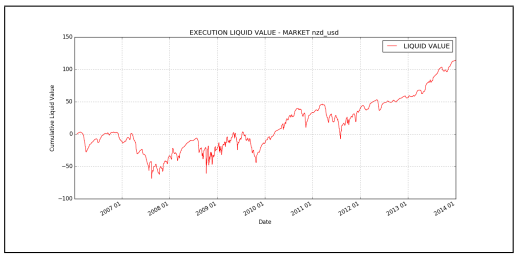


Figure 56: Results of the Version 4 - NZD/USD

USD/CHF

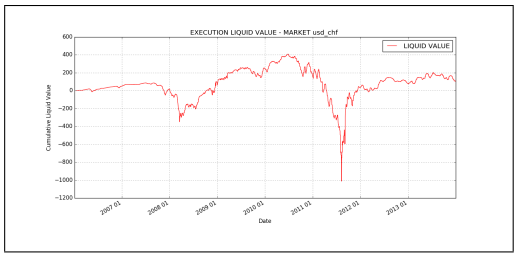


Figure 57: Results of the Version 1 - USD/CHF

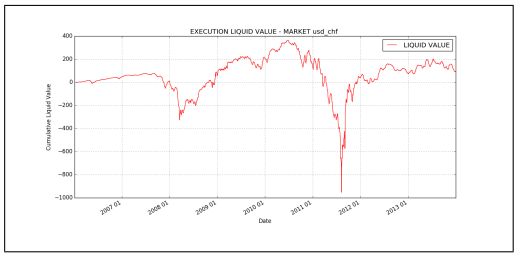


Figure 58: Results of the Version 2 - USD/CHF

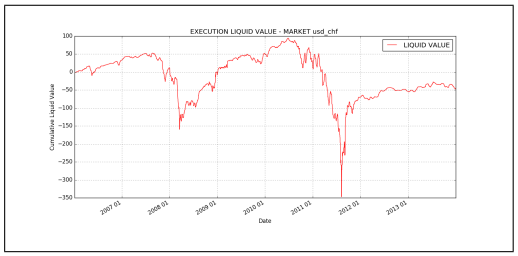


Figure 59: Results of the Version 3 - USD/CHF

Figure 60: Results of the Version 4 - USD/CHF

XAG/USD

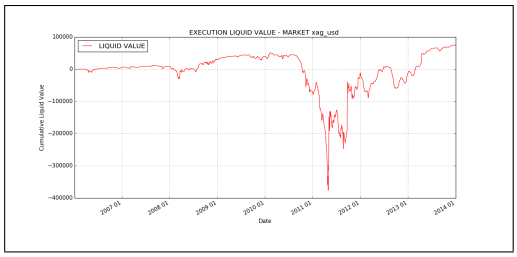


Figure 61: Results of the Version 1 - XAG/USD

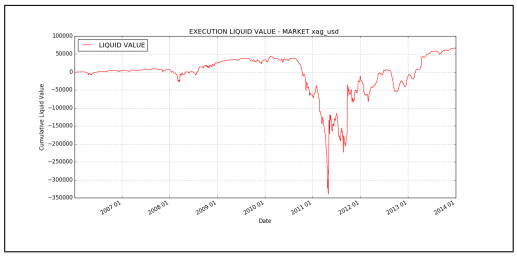


Figure 62: Results of the Version 2 - XAG/USD

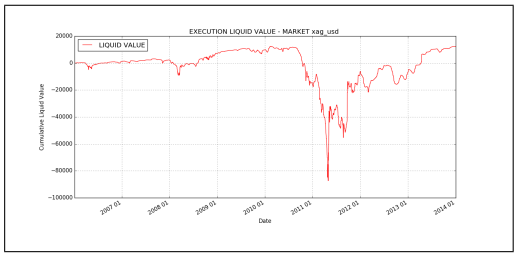


Figure 63: Results of the Version 3 - XAG/USD

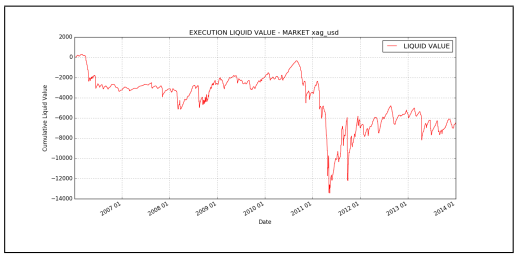


Figure 64: Results of the Version 4 - XAG/USD

XAU/USD

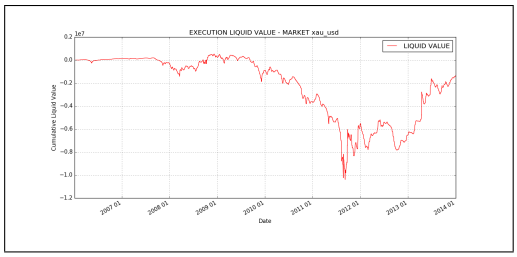


Figure 65: Results of the Version 1 - XAU/USD

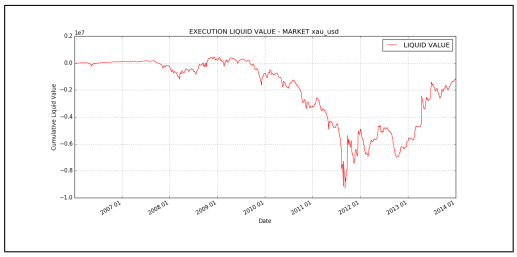


Figure 66: Results of the Version 2 - XAU/USD

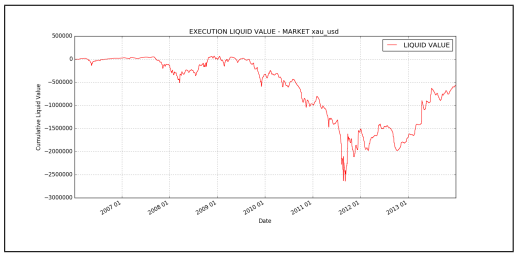


Figure 67: Results of the Version 3 - XAU/USD

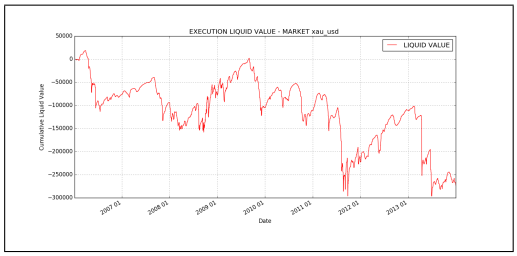


Figure 68: Results of the Version 4 - XAU/USD

How to set up the project

Install Python and pip

.0.1 Windows

Go to https://www.python.org/downloads/windows/ and download the version 3.7.2. Select either Windows x86-64 executable installer for 64-bit or Windows x86 executable installer for 32-bit. Once you have downloaded an installer, simply run the installer by double clicking. Follow installer instructions. You can check if the installation worked by typing the terminal command:

```
$ python -V
```

The output should return:

```
Python 3.7.2
```

Once Python is installed, install pip downloading the file 'get-pip.py' from the website https://bootstrap.pypa.io/get-pip.py to a folder on your computer. Open a command prompt and navigate to the folder containing get-pip.py. Run the following command:

```
$ python get-pip.py
```

You can verify that Pip was installed correctly by opening a command prompt and entering the following command:

```
$ pip -v
```

You should get an output similar to the following:

```
pip 18.0 from c:/users/administrator/appdata/local/programs/python/python37/lib/site-packages/pip (python 3.7)
```

.0.2 Mac OS X

Install brew by pasting the following in a macOS Terminal prompt.

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Once brew is installed:

```
$ brew install python3
```

You can check if the installation worked by typing the terminal command:

```
$ python3 –version
```

The output should return:

```
Python 3.7.2
```

Once Python is installed, install pip by typing:

```
$ sudo easy_install pip
```

Type the next to check if the installation worked:

```
$ pip –version
```

The output should be like:

```
pip 19.1.1 from /Users/name/virtualenvironment/interface/lib/python3.7/site-packages/pip (python 3.7)
```

.0.3 Ubuntu 16.10 or newer

Open the terminal and type the following commands:

```
$ sudo apt-get update
```

```
$ sudo apt-get install python3.6
```

You can check if the installation worked by typing:

```
$ python –Version
```

The output should return:

```
Python 3.7.2
```

Once Python is installed, install pip by typing:

```
$ sudo apt update
```

```
$ sudo apt install python3-pip
```

Once the installation is completed, verify the installation by checking the pip version:

```
$ pip3 –version
```

The version number may vary, but it will look something like this:

```
$ pip 9.0.1 from /usr/lib/python3/dist-packages (python 3.6)
```

Set up a virtual environment

.0.4 Windows

Type the following commands:

```
$ virtualenv my_venv
```

```
$ pip install virtualenv
```

```
$ my_venv\Scripts\activate
```

.0.5 Mac OS X

Type the following commands:

```
$ pip install virtualenv
```

```
$ virtualenv my_venv
```

```
$ source my_venv/bin/activate
```

.0.6 Ubuntu 16.04

Type the following commands:

```
$ sudo apt-get update
```

```
$ sudo pip3 install virtualenv
```

```
$ virtualenv venv
```

```
$ source venv/bin/activate
```

Install the requirements and run the program

Open a terminal and go to the folder where you have saved the Final Degree Project code files. If you have not downloaded the project, you can download and unzip it in the folder you wish from the next GitHub repository:

```
https://github.com/luis3zc/TFG/archive/master.zip
```

Install the requirements by typing:

```
$ pip install requirements.txt
```

To run the program type:

```
$ python3 Main.py
```

Bibliography

[1] RialtoTrade, "Beginnings of algorithmic trading." https://medium.com/rialto-ai/beginnings-of-algorithmic-trading-19eccce902a1, 2018.

[2] QuantInsti, "History of algorithmic trading, hft and news based trading." https://www.quantinsti.com/blog/history-algorithmic-trading-hft, 2015.

[3] Wikipedia, "Algorithmic trading." https://en.wikipedia.org/wiki/Algorithmic_trading, 2019.

[4] U.S. Presidential Task Force on Market Mechanisms, "Report of the Presidential Task Force on Market Mechanisms : submitted to The President of the United States, The Secretary of the Treasury, and The Chairman of the Federal Reserve Board." https://archive.org/details/reportofpresiden01unit, 1988.

[5] U.S. Securities and Exchange Commission, "Commission Notice: Decimals Implementation Plan for the Equities and Options Markets." https://www.sec.gov/rules/other/decimalp.htm, 2000.

[6] R. Das, J. Hanson, J. Kephart, and G. Tesauro, "Agent-human interactions in the continuous double auction," *IJCAI International Joint Conference on Artificial Intelligence*, 05 2001.

[7] J. Chen, "Trading Psychology." https://www.investopedia.com/terms/t/trading-psychology.asp, 2018.

[8] J. Montier, *The Little Book of Behavioral Investing: How not to be your own worst enemy*. John Wiley & Sons, Ltd., 2010.

[9] Wikipedia, "High-frequency trading." https://en.wikipedia.org/wiki/High-frequency_trading, 2019.

[10] J. Treanor, "The 2010 'flash crash': how it unfolded," *The Guardian*, 2015.

[11] F. Fernández, "The foreign exchange market and its participants." https://www.bbva.com/en/foreign-exchange-market-participants/, 2017.

[12] Bank for International Settlements, Monetary and Economic Department, "Foreign exchange turnover in April 2016." https://www.bis.org/publ/rpfx16.htm, 2016.

[13] G. Protonotarios, "The microstructure of forex market speculation." https://forex-rebates.com/index.php/forex-tips/speculation, 2017.

[14] History.com, "Great recession timeline." https://www.history.com/topics/21st-century/great-recession-timeline, 2018.

[15] Swiss National Bank, "Swiss National Bank sets minimum exchange rate at CHF 1.20 per euro." https://www.snb.ch/en/mmr/reference/pre_20110906/source/pre_20110906.en.pdf, 2011.

[16] G. Dorgan, "SNB Losses: 1.85 Billion Francs in Just One Day, 231 Francs, 250$ per Inhabitants." https://snbchf.com/2012/12/snb-losses-dec18/, 2012.

[17] M. Aloud, E. Tsang, R. Olsen, and A. Dupuis, "A Directional-Change Event Approach for Studying Financial Time Series," *Economics*, vol. 6, pp. 2–8, 2012.

[18] B. Mandelbrot and H. Taylor, "On the Distribution of Stock Price Differences," *Operations Research*, vol. 15, no. 6, 1967.

[19] J. Stock, "Estimating Continuous-Time Processes Subject to Time Deformation: An Application to Postwar U.S. GNP," *Journal of the American Statistical Association*, vol. 83, no. 401, 1988.

[20] U. Muller, M. Dacorogna, R. Davé, O. Pictet, R. Olsen, and J. Ward, "Fractals and Intrinsic Time - A Challenge to Econometricians.," *Olsen and Associates*, vol. 1, no. 2, 1997.

[21] D. Guillaume, M. Dacorogna, R. Davé, U. Müller, R. Olsen, and O. Pictet, "From the bird's eye to the microscope: A survey of new stylized facts of the intra-daily foreign exchange markets," *Finance Stoch*, vol. 1, no. 2, 1997.

[22] A. Golub, J. Glattfelder, and R. Olsen, "The Alpha Engine: Designing an Automated Trading Algorithm." https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2951348, 2017.

[23] "The alpha engine: Designing an automated trading algorithm code." https://github.com/AntonVonGolub/Code/blob/master/ code.java, 2017.

[24] G. J., D. A., and O. R., "Patterns in high-frequency FX data: Discovery of 12 empirical scaling laws," *Quantitative Finance*, vol. 11, no. 4, 2008.

[25] Wikipedia, "Power law." https://en.wikipedia.org/wiki/Power_law, 2019.

[26] A. Dupuis and R. Olsen, "High Frequency Finance: Using Scaling Laws to Build Trading Models," *Handbook of Exchange Rates*, 2012.

[27] P. S., *The Evaluation of the Trend-Following Directional Change with the Trailing Stop and Major-Trend-Adjusted Strategies on Algorithmic Trading in the Foreign Exchange Markets*. PhD thesis, University of Essex, 2015.

[28] A. Golub, G. D. A. ., Chliamovitch, and B. Chopar, "Multi-scale representation of high frequency market liquidity," *Algorithmic Finance*, 2016.

[29] Wikipedia, "Entropy (information theory)." https://en.wikipedia.org/wiki/Entropy_(information_t 2019.

[30] T. Cover and J. Thomas, *Elements of Information Theory*. New York, NY, USA: Wiley-Interscience, 1991.

[31] P. Algoet and T. Cover, "A sandwich proof of the shannon-mcmillan-breiman theorem," *The Annals of Probability*, vol. 16, no. 2, pp. 899–909, 1988.

[32] H. Pfister, J. Soriaga, and P. Siegel, *On the achievable information rates of finite state ISI channels*, vol. 5. GLOBECOM'01. IEEE Global Telecommunications Conference, 2001.

[33] L. Cam, "The central limit theorem around 1935," *Statistical Science*, vol. 1, pp. 78–91, 1986.

[34] A. Golub, G. Chliamovitch, A. Dupuis, and B. Chopard, "Multi-scale Representation of High Frequency Market Liquidity." https://arxiv.org/pdf/1402.2198.pdf, 2014.

[35] J. Golub A., Glattfelder, V. Pertov, and R. Olsen, "Waiting Times and Number of Directional Changes in Intrinsic Time Framework.," *Algorithmic Finance*, 2017.

[36] J. Chen, "Stop-Loss Order." https://www.investopedia.com/terms/s/stop-lossorder.asp, 2018.

[37] C. Mitchel, "Trailing Stop Definition and Uses." https://www.investopedia.com/terms/t/trailingstop.asp, 2019.

[38] C. Mitchel, "Breakeven Point (BEP) Definition and Examples." https://www.investopedia.com/terms/b/breakevenpoint.asp, 2019.

[39] W. Kenton, "Risk/Reward Ratio Definition." https://www.investopedia.com/terms/r/riskrewardra 2019.

[40] "The alpha engine: Discussion page." https://github.com/AntonVonGolub/Code/issues/11, 2018.

[41] P. Kruchten, "The 4+1 view model of architecture," *IEEE Softw.*, vol. 12, pp. 42–50, Nov. 1995.

[42] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[43] FXCM, "fxcmpy algorithmic trading with fxcm." https://fxcmpy.tpq.io/index.html#, 2019.

[44] Wikipedia, "Acid." https://en.wikipedia.org/wiki/ACID, 2019.

List of Figures

List of Tables

List of Algorithms

Index