Department of Informatics
King's College London
United Kingdom

7CCSMPRJ MSc Project

TESTING THE LIBRA MECHANISM IN AN AGENT BASED MODEL

ABSTRACT

Major financial exchanges often consider how to improve their matching protocols to make them fairer (disincentivising undesired behaviour) or more efficient (computationally or in terms of complexity), whilst not changing the underlying market dynamic by so much that the order flow is driven elsewhere.

In the recent literature, a novel matching mechanism (LIBRA) was suggested – which solves a well-defined 'temporal fairness' issue, and this algorithm has been tested in a limited exchange setting. However, this test was conducted on a small exchange, not comparable to the major global exchanges through which most of the orders flow (World Federation of Exchanges, 2021). Furthermore, these exchanges are often hesitant to use untested algorithms in live markets given the sums of money at play – notwithstanding the importance of these systems remaining functional for global financial markets to function. To address this gap in the literature, I developed a flexible agent-based model platform that accepts LIBRA as a matching mechanism.

One of the largest concerns for these major exchanges (e.g., LSE, NYSE, NASDAQ) is how a new mechanism will affect the liquidity of a given asset – as this tends to be correlated with providing better prices than competitors (Amihud & Mendelson, 1988). With this in mind, the bid/offer spread whilst using the LIBRA mechanism is compared to the prevailing 'First Come First Served' (FCFS) standard[1]. Using the software, it is found that LIBRA does not materially change the spread – and therefore would be suitable for deployment in major exchanges.

The platform developed has a functional GUI (to improve ease of use) and allows for the addition of future matching mechanisms to increase the usefulness of the software (e.g.,

[1] It is recognised that some exchanges use a different auction policy to set opening/closing prices (Hinterleitner, Leopold-Wildburger, Mestel, & Palan, 2015). For the purposes of this paper, these exceptions are generally ignored.

Frequent Batch Auctions (Budish, Che, Kojima, & Milgrom, 2013)) – so that it can be used for similar studies in the future.

NOMENCLATURE

E – an exchange
op – a market opportunity
ϵ – a small delay
U_i – the time it takes from a participant submitting an order until it arrives at the exchange
L – the length of a matching interval (i.e., the buffer time)
$\delta\tau$ – a small amount of time

P – the set of participants (e.g., in a buffer)
P_i – a given participant i
$P_i[]$ – the list of orders that a participant has placed into a specific buffer

O_s – the set of sell orders in the orderbook
O_b – the set of buy orders in the orderbook

o – an order
o_p – the price of a given order o
o_s – the side (buy/sell) of a given order o
o_t – the type (market/limit/etc.) of a given order o
o_u – the unique code of the trader who submitted a given order o
o_i – the unique code of the asset for which a given order o applies

b_i – a given buffer

$O(\cdot)$ – big O notation, referring to computational complexity (Bae, 2019)
$\max(\cdot)$ – the maximum function
$\min(\cdot)$ – the minimum function
$\mathrm{a(t)}$ – the (log of the) best ask price at time t
$\mathrm{b(t)}$ – the (log of the) best bid price at time t

γ – a random delay enforced on trades arriving at an exchange
λ – the probability a trader is picked in Chiarelli-Iori

t – time
τ – an amount of time

σ – the number of shares in an order
μ – the arrival rate of market orders
α – the arrival rate of limit orders
η – the decay rate of orders

$N_{O,i}$ – the number of outstanding orders participant i has

$x \ll y$ – x is **much** smaller than y

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Through both ancient and modern history, humans have been fascinated by the subject of trade, and how their gains can be maximised from it. In the literature, archaeologists have described how long-distance, commercialised trade was occurring in Mesopotamian society as far back as 3000 years ago, in commodities such as gold, silver, copper, grains and wool (which echoes modern commodities markets! (Lioudis & Silberstein, 2021)). Going even further back, recent research has suggested that trade may have been occurring as far back as 300,000 years ago (Polianskaya, 2018)!

Originally, this trade was largely in the form of bartering and traditional trade – goods would be exchanged for each other depending on need. However, as currencies were developed as a mode of storing wealth, these started to be used instead (Bernholz, 2003). As the world became increasingly globalised, and interest in investing picked up (Etheridge, 2021), we shifted to a world in which trade is done through exchanges. One famous early example is the Holland Stock Exchange (National Museum of Australia, 2021), which exhibited one of the first notable examples of a bubble in its tulip futures market in the mid 1600s (Corporate Finance Institute, 2021). Then, as we moved into the electronic age, virtually all exchange business went online[2], bringing buyers and sellers together virtually in micro (or even nano) seconds (Lacasse, 2012).

[2] Barring some (extremely limited) pit trading, that remains for historical and theatrical reasons, e.g., Eurodollar options at the Chicago Mercantile Exchange (CME) (Reuters, 2021).

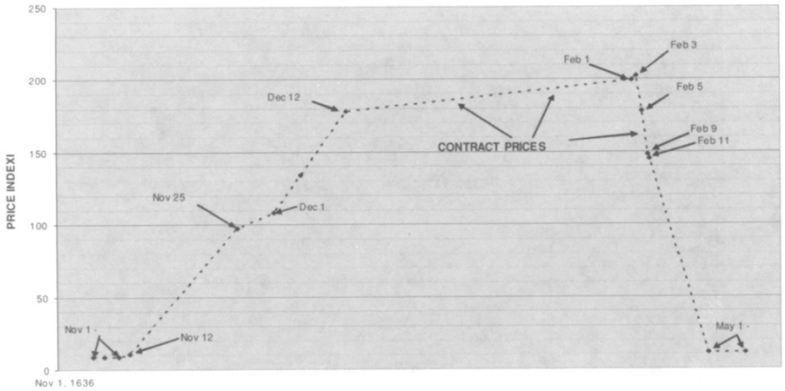

Figure 1: Graph of the Tulip Bubble in the 1600s (Thompson, 2007).

One issue that this introduced is that exchanges could now, more easily than before, change and refine their matching mechanism – the method of bringing buyers and sellers together. One of the most used mechanisms is 'First Come First Served' (FCFS) (Mavroudis & Melton, 2019). This places a preference on matching buyers and sellers whose orders arrive (to the exchange) chronologically earlier than others at a given price level. Naturally, this can create some undesirable effects – perhaps most importantly, it can result in traders who place orders first being further back in the priority queue than traders they have reacted faster than (perhaps resulting from network lag) – this issue is codified by Mavroudis and Melton as 'Temporal Fairness'. This problem has increased in magnitude in correlation with the growth of high-frequency and ultra-high-frequency trading. Several approaches to the problem have been suggested: two of which I will discuss in detail, and one of which I have successfully implemented in the agent-based model. The core quantitative focus of this project is the LIBRA mechanism (Mavroudis & Melton, 2019); however, another important solution is the Frequent Batch Auction (FBA) (Budish, Che, Kojima, & Milgrom, 2013), which I will qualitatively discuss – and had this project been over a longer duration, I would also have implemented in the software.

There are also other issues associated with the prevailing FCFS standard. First, it can incentivise traders to place orders which they have little current intention of holding to execution (they simply value first place in the queue at each price). Second, it has been a factor in the creating the oft-mentioned 'technological arms race' (Aldridge, 2013) where firms pay vast sums to either lease office space geographically close to a given exchange (e.g., (Colt, 2011)) or choose to be co-located (where the option is offered). Co-location refers to having hardware physically on-site at the exchange, with the minimal available access times; the number of exchanges offering such opportunities has been increasing in recent years (Goldman, 2012) with the growth of algorithmic ('algo') funds, who often require near-immediate access times for their strategies to be most profitable (Duhigg, 2009).

1.1 Aims and Objectives

Formally, the prime aim of the project is to expand the knowledge and understanding of the LIBRA mechanism (notably it has not been tested in an appropriate environment), and to compare it to other existing mechanisms. To do this, I will apply existing techniques in agent-based modelling to build an appropriate synthetic environment which is based on major financial exchanges, and their established mechanisms.

Further, there is a secondary aim to have the code continue to be used by similar projects in the future, perhaps as the field of study is advanced – often such work remains dormant for long periods following its conclusion. The approach to meeting this aim has two strands: first, ensuring that the software will accept other mechanisms. Second, the user experience should be pleasant enough that there is not a steep barrier to entry.

To meet these aims, a series of objectives were pursued as follows (these are elaborated on in chapter 3):

- Build a base platform including a central exchange, orderbook(s) and main game loop.
- Build traders on this platform which will give and take liquidity depending on their type.
- Test the model and refine it using data from real markets.
- Build a suitable user interface on top of the simulation.
- Generate data on the spread for both LIBRA and FCFS.
- Quantitatively compare the two mechanisms, and perhaps provide a qualitative comparison to FBA.

1.1.1 Structure

The following report is structured as follows:

- Background and Literature Survey: a brief overview of key papers used during the project.
- Background Theories: in this chapter I consider the impact of high-frequency trading on marketplaces and discuss some of the negative consequences it causes. I address previous findings in the field and lay the groundwork off which the simulation will be built.
- Objectives, Specifications and Design: here I expand on the aims and objectives discussed above. I then go on to lay out some specifications for the software and highlight some design features in the simulation.
- Methodology and Implementation: in this section I discuss some background about Agent-Based Modelling, describe and evaluate some market models and then define the one we will use. Following this, I describe how some features from the specification and method were implemented.
- Results, Analysis and Evaluation: here, I conduct a hypothesis test to determine whether LIBRA significantly affects the spread when compared to FCFS.
- Legal, Social, Ethical and Professional Issues: this short chapter discusses the BCS Codes of Conduct and Good Practice, and how the project has met these professional standards.
- Conclusion: Here I sum up the findings of the report and discuss some thoughts on the future directions of the work, and the field at large.

1.2 Background and Literature Survey

1.2.1 Libra: Fair Order-Matching for Electronic Financial Exchanges

As I mentioned earlier, most of the time working on the project surrounded the paper which introduced the LIBRA mechanism (Mavroudis & Melton, 2019). Within this publication, Mavroudis and Melton codify the (late-arrival) fairness condition I described earlier and describe the sufficient condition to avoid it as 'temporal fairness'. Following this, they design a matching mechanism (LIBRA) which (in-theory) meets the condition discussed. They then go on to test the mechanism in a limited foreign-exchange environment, hosted on Refinitiv (the organisation one of the researchers works for). They conclude with the results that based on their theoretical analysis and experiments LIBRA has a limited market impact when compared to other state-of-the-art matching techniques.

One strength to note is that, in this paper, the level of mathematics and notation is kept relatively low-level. Further, diagrams are used regularly to express points that the authors are trying to convey (e.g., 'temporal fairness'). This means that the paper is more approachable and readable for those who are not subject matter experts (SMEs) in the field.

1.2.2 An Agent-Based Simulation of Double-Auction Markets

Another piece of literature which my project relies and builds upon is the thesis written by Guo (Guo, 2005). This discusses how an agent-based model can be built and later used in experiments. Guo prefers to use a thread and socket-based approach in his research, whereby each trader is created as a separate program, then these programs interface over the network with a central exchange. This central party then maintains a thread for each connection, over which traders are updated on the market and can place orders. Although this sounds like a suitable approach, it vastly increases the software complexity, and as we have no intentions of using the code from this project in live, classroom-based experiments, it would be a waste of resources and time to develop. However, it is recognised that the approach is perfect for such environments.

Helpfully, the paper has an extensive section on future work and describes three models to simulate order flow: the Chiarelli-Iori model (Chiarella & Iori, 2002), the Das model (Das, 2003) and the SFGK model (Smith, Farmer, Gillemot, & Krishnamurthy, 2003). Guo helpfully then provides a summary of each and describes some of the underlying market assumptions, in addition to highlighting some of the strengths and drawbacks of each.

1.2.3 The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response

In their work, Mavroudis and Melton also discuss alternative matching mechanisms, one of which is the Frequent Batch Auction (Budish, Che, Kojima, & Milgrom, 2013). The paper from which this mechanism originates is extensive, but seminal, and highlights how the FBA approach solves some flaws identified in the continuous limit order book. They also discuss

some other approaches, demonstrating knowledge of the field, such as Tobin taxes (Tobin, 1978), bans to HFT (suggested by a previous winner of the Nobel Prize in Economics - (Phillips, 2011)) and random message delays (Harris, 2012). To each, they then go on to state their qualms, and reasons to prefer another approach.

In all, I believe that the detail contained in the paper lends credence to the researchers' conclusions. However, it must be noted that having a shorter final publication would have increased the accessibility of the work, and perhaps increased its utility to most readers.

2 BACKGROUND THEORIES

According to the ESMA (European Securities and Markets Authority), high-frequency trading now accounts for as much as 76% of all flow[3] in European equity markets (Breckenfelder, 2020). As the prevalence of these strategies has increased, so has the amount of work concerning their impact on the market, and on ways that these impacts can be mitigated. In this section, I discuss a selection of market issues in which high-frequency trading has been named as a potential culprit, then go on to discuss some of the solutions to these issues, including work that will be built upon in my software simulation.

2.1 Issues Identified

2.1.1 'Race to the Exchange'

The first issue that I will discuss is the oft-mentioned 'Race to the Exchange'. As technology has continued to develop, high-frequency traders have looked for quicker and quicker access times at major exchanges. This is because these strategies 'as a prerequisite, … need to rely on high-speed access to markets' (Gomber, Arndt, Lutat, & Uhle, 2011). Perhaps one of the greatest indicators of this race is the growth of data centres (one use of which is co-location (ice, 2021)) at exchanges; whilst the old New York Stock Exchange (NYSE) building occupied 46,000 square feet, the new NYSE data centre in New Jersey is approximately nine times larger, at 400,000 square feet (Lewis, 2014). Another (potentially less-expensive) way

[3] Measured by number of orders.

that high-frequency traders can gain (reasonably) fast access to an exchange's servers is via third-party proximity services, such as those offered by QuantHouse (QuantHouse, 2021).

Despite it being suggested by industry leaders as early as 2015 that 'the race to zero is almost over' (Brown, 2015), this issue continues to plague the market – with traders looking for marginal access time advantages. Why? Well, in the literature it has been suggested that (as of July 2021) over 20% of trading volume takes place in 'races' – where the arbitrage opportunities are sufficiently mechanical so that whichever firm takes liquidity first will realise the profits (Aquilina, Budish, & O'Neill, 2021). In the cited paper, researchers find that this 'latency arbitrage' accounts for 31% of all price impact, and that minimising it would reduce the cost of liquidity for investors by as much as 17%.

Whether such a race is socially desirable is a topic that has been well studied by academics in the recent literature (Budish, Lee, & Shim, 2019), (Harris, What to do about high-frequency trading, 2013), (Daian, et al., 2019) and even one that has seen attention from the general public (Lewis, 2014). Within these publications, this race is often deemed to be wasteful, and compared to the prisoner's dilemma – that is, all participants would benefit if markets were redesigned to de-emphasize access speed. However, it is not a closed discussion: some are of the opinion that this race is, in and of itself, a fundamental part of the competition. The commissioners who objected to the 3ms speedbump at the Intercontinental Exchange (ice) stated that: 'Risk (and reward) move at the speed of information. Those that invent, and invest in, faster information transmission technologies to capitalize on market dislocations reap the profits of their advantage.' (Quintenz, 2019).

2.1.2 Temporal Fairness

In a traditionally strict sense, the FCFS continuous limit order book (CLOB) is (at least theoretically) fair, whereby the fastest response to a given event is given the first opportunity to obtain the best bid/ask prices or to place a new order (Harris, What to do about high-frequency trading, 2013). However, Mavroudis and Melton highlight that 'this assumes that assumes that a market's *implementation* meets its *specification* directly' (Mavroudis & Melton, 2019). In their work, the researchers note that modern exchanges are often unable to meet this specification exactly. They highlight that 'jitter' arises from modern hardware (which is optimised for maximum performance and hence does not guarantee constant runtime), the large distributed systems which exchange platforms run on are so hard to balance that some degree of asymmetric delay is inevitable, and that a selection of technical market manipulation techniques may also result in the strict fairness condition being broken.

To overcome these issues, the researchers propose a new way to define fairness, 'temporal fairness'. This mechanism also mitigates the ongoing race to the exchange by introducing the potential to buffer orders together. Formally, given two market participants P_1 and P_2, a market opportunity op and an exchange E, the exchange is defined as being temporally fair if among all pairs of participants on the exchange the probability that a slower participant succeeds at capturing op at the expense of a faster participant is at most 0.5.

Although this may seem to be a trivial condition to achieve, exchanges have in the past been fined for allowing slower actors to overtake quicker ones, be that by offering 'preferential access' channels to certain participants (Securities and Exchange Board of India, 2019), or by offering access early to specific groups of traders (Osipovich, 2017). Furthermore, some

violations of temporal fairness may by unavoidable when using traditional exchanges. For example, there may be latency when distributing data to market participants. As can be seen in the figure below (Figure 2), there may be an ϵ introduced when submitting orders (e.g., if a unicast transmission system is used). However, even if this is mitigated (e.g., we assume that the TCP/UDP packets are sent simultaneously using a one-shot multicast system – ignoring the concerns raised in (Klocking, Maihofer, & Rothermel, 2001)), $U_2 - U_1$ can still be greater than zero – perhaps caused by marginal differences in cable lengths or congested routers – which may potentially result in fast-reacting participants being disadvantaged against slower-reacting ones.

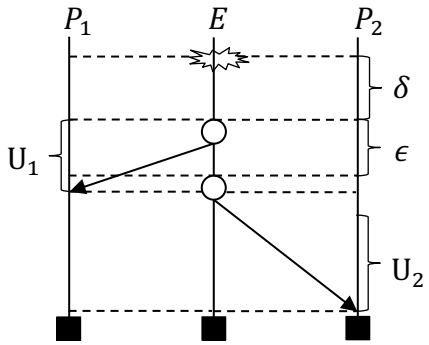

Figure 2: Diagram showing the potential sources of unfairness when broadcasting and receiving messages.

2.1.3 Impact on Market Volatility

Another common concern of high frequency trading (which is not analysed as in-depth in this report), is its potential to affect market volatility. The literature on this area is often conflicting, depending on the parameters of the study. Some papers conclude that volatility is lowered by high-frequency traders (Brogaard, 2010) (who, in theory, provide more liquidity in the marketplace), whereas others provide convincing statistics to support the argument that volatility is worsened by HFT (Caivanoo, 2015). Direct comparison of such studies may be challenging due to the impact external market factors can have on volatility, although some research has attempted to explain the reasons for the different conclusions (Sornette & von der Becke, 2011).

One highly visible illustration of the impact that HFT firms can have is a so-called 'flash crash'. One example occurred on May 6th, 2010, when Navinder Sarao (a trader from the UK,

affectionately referred to by some as the 'Hound of Hounslow'[4]) deliberately spoofed several large trades on the E-mini[5] futures market (Verity, Lawrie, & Eleanor, 2020). The resulting sell-off and withdrawal of liquidity by HFT firms accelerated the following crash, and within 20 minutes, US securities markets had lost over 1,000,000,000,000 USD in value, over 9% of their total (see Figure 3). Within 2 hours, markets had recovered to their pre-crash levels (Poirier, 2012).

[4] In reference to Jordan Belfort, the 'Wolf of Wall Street' (Anderson, 2021).
[5] An 'E-mini' is a commonly traded futures contract with a notional value of fifty times the S&P 500 stock market index (Chen & Scott, 2021).

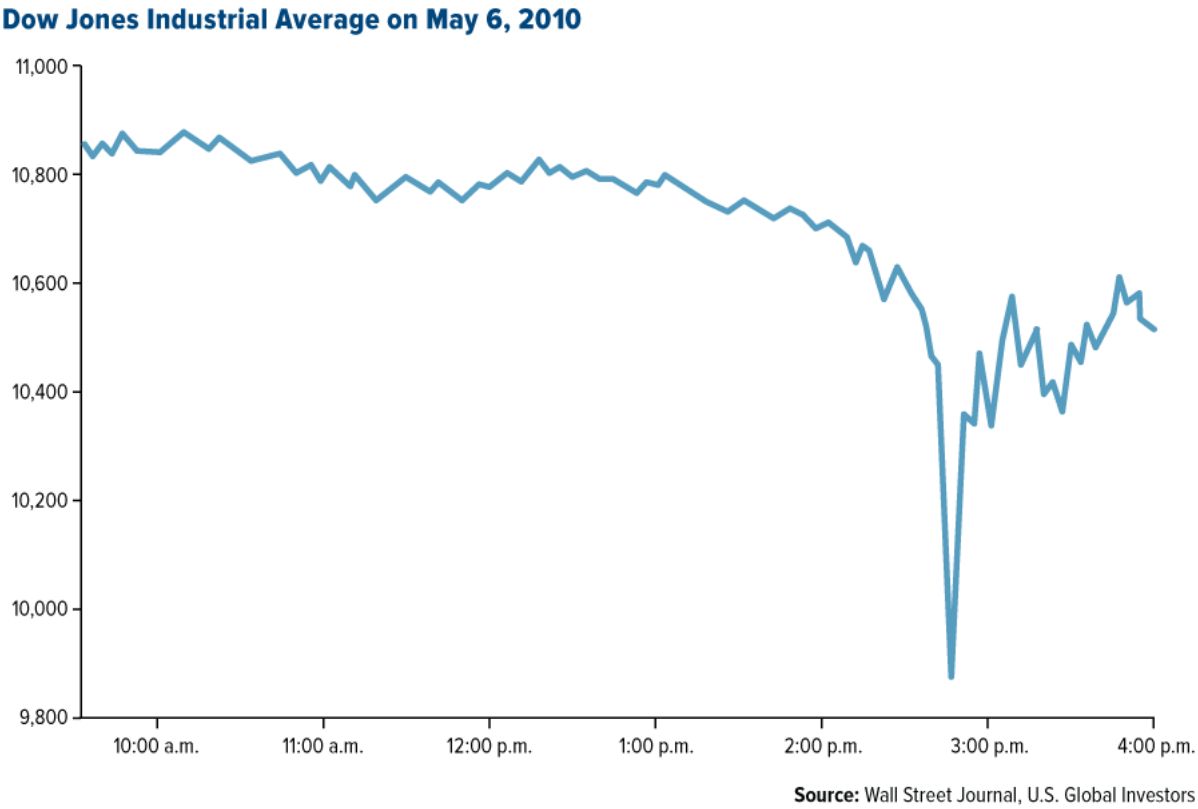

Figure 3: Graph of the May 6th, 2010, Flash Crash (Holmes, 2015).

2.2 Mechanisms

To address the issues described above, several mechanisms have been proposed as alternatives to the FCFS standard. Researchers have noted that it is important that firms do not ignore any of these issues when considering new mechanisms, especially ones which they do not have the ability to control – we 'need to deal with all of a market's complications, not just its principal features' when designing a new mechanism (Roth, 2002). In this section, I discuss how FCFS works, and then compare it to LIBRA (Mavroudis & Melton, 2019) and FBA (Budish, Che, Kojima, & Milgrom, 2013).

2.2.1 First Come First Served (FCFS)

This seminal mechanism is the simplest of the three that I will discuss. Functionally, it works by accepting orders straight into the classic continuous limit order book (CLOB). If an incoming order o is a buy (sell) market order, and there is enough liquidity available from the offer (ask) limit orders (i.e., enough shares available to buy (sell) from the stored orders), then it will match the incoming order instantly against the stored, already existing ones. If the order is a limit order, there are two options. Either it is a marketable limit order (i.e., the limit buy (sell) price - o_p exceeds the best available ask (bid) - $\max(o_p | o \in O_s)$), in which case the order will be treated as a market order, until o is either fully satisfied or become a non-marketable limit order (i.e., the limit buy (sell) price no longer exceeds the best available ask

(bid)). Non-marketable orders are simply added and stored to the orderbook, changing the best bid/ask price if appropriate.

This mechanism guarantees that the message which is received the fastest (which, in much of the theory, is assumed to be the one which is sent first) after an event will capture the associated trading opportunity.

2.2.2 LIBRA

The first suggested alternative to FCFS is LIBRA (Mavroudis & Melton, 2019). This mechanism is based on the premise of stochastically equalizing (i.e., equalizing in matching probability) orders which are within the error buffer defined by the exchange. Functionally, it works by collecting orders in buffers (generally according to price), before forwarding them into the same matching engine that is used in FCFS, with the same underlying orderbook.

Each order o is required to have five qualities when using LIBRA: a unique code declaring the party who submitted the order o_u, a unique code declaring the asset to be traded o_i, a type of order o_t (Market, Limit, Cancel, IOC – 'immediate or cancel'), whether the order is on the buy or sell side o_s, and a limit order price o_p.

An order o follows a path through the exchange as follows: when o arrives, the exchange runs Algorithm 1 (see Figure 4). If $o_t = \text{cancel}$, then the order is immediately forwarded into the matching engine. Otherwise, the order is placed into a buffer b_i, where orders are grouped based on two factors, first, whether they are 'marketable' (Peterson & Sirri, 2002) (i.e., they cross the spread as defined above), and if they are non-marketable, then based on their price o_p. Marketable orders are stored in a two common buffers, one for each side of the market, whereas other orders are stored in a buffer according to their price level. If a buffer was previously empty, then a timer now starts counting to a predetermined amount (the buffer length).

```
instrument = o_i
side = o_s
type = o_t

if type = cancel:
        Forward(o)
        Return()

elif side = buy && order is marketable:
        b = marketable buy buffer

elif side = sell && order is marketable:
        b = marketable sell buffer

elif type != 'Immediate or Cancel':
        price = o_p
        b = price specific buffer

if b contains orders:
        append o to the buffer

else:
        create a buffer object b which contains o
        start the buffer timer
```

Figure 4: LIBRA Algorithm 1.

Once the timer finishes, Algorithm 2 is run on the buffer (see Figure 5). Here, orders are first placed in lists according to each participant $P_i[]$, then each list is sorted according to arrival time. Following this, a random permutation is found on the set P of participants, and the participants are then arranged in this new order. Finally, orders are iteratively popped from each participant, cycling through the permutation until all lists are empty.

```
for each order in the buffer:
        append it to the correct P_i[ ]

for each user in the buffer:
        sort their orders by arrival time

R = randomised permutation of the list of participants

while orders remain in the buffer:
        for each user in R:
                if an order remains in this user's P_i[ ] then pop it and forward it
```

Figure 5: LIBRA Algorithm 2.

This mechanism, does, under non-faulty conditions, satisfy the 'temporal fairness' condition mentioned above. However, it also has the potential to neutralise participants who legitimately act (a small amount) faster than their competitors. In the literature, is has been suggested that this equalisation could be socially desirable (Budish, Lee, & Shim, 2019).

2.2.3 Frequent Batch Auctions (FBA)

The third mechanism to discuss (and one which has received a fair amount of attention in the literature) is the frequent batch auction (Budish, Che, Kojima, & Milgrom, 2013). This mechanism suggests discretizing the day into short matching intervals (each of length L), as opposed to matching orders in (near[6]) continuous time. At the end of each discrete interval, a call auction is performed[7]. In their research, Budish et al. suggest that an appropriate magnitude for L is around a tenth of a second.

Assuming that trading triggers are generated uniformly throughout the day, the probability of two traders P_1 and P_2 (who suffers from network latency of $\delta\tau$ compared to P_1), who react simultaneously to the same trigger op, ending up in the same interval (and thus the same auction) is $\frac{L-\delta\tau}{L}$. Hence, the probability of P_2 winning the opportunity is $\frac{L-\delta\tau}{L} \cdot \frac{1}{2}$. Notably, this requires $\delta\tau \ll L$ for P_2 to have a (near) 50% chance of winning an opportunity – hardly a desirable condition when minimal market impact is a key aim. As an example, assume that P_1 takes 2ms to send a message to the exchange, whereas P_2 takes 3ms. Here, $\delta\tau$ is 1ms. Now assume that the length of each interval is 2ms. Then the probability of P_2 winning the opportunity is $\frac{2-1}{2} \cdot \frac{1}{2} = \frac{1}{4}$.

Comparing this to LIBRA – given the same two traders, with the same network latency, by making the timer count to 1ms we give P_2 the necessary time to submit their order within the network lag tolerance. Further, if two traders place an order in the same buffer then they have an equal probability of winning the opportunity (as the traders are randomly permutated

[6] Small low-level unavoidable hardware delays can mean that matching in continuous time in a mathematical sense is not possible.

[7] A call auction accumulates all orders that have arrived during the length of the auction, then matches all buy and sell orders that it can using a single clearing price (Hayes, 2021). This mechanism is often also used to set opening and closing prices at major exchanges (Li, Luo, & Zhou, 2021).

before pulling out orders). Hence, with only a 1ms delay we give P_2 a $\frac{1}{2}$ probability of being matched first. This demonstrates one of the major advantages of LIBRA over FBA.

2.3 Alternative Solutions

Researchers have not only suggested changing the fundamental market mechanism to prevent the race to the exchange – they have also suggested a wide range of (generally) simpler solutions, such as implementing taxes on trades (Tobin, 1978) or implementing a random message delay on traders (Harris, 2012). In this section I recognise, and briefly discuss, some of these alternatives.

2.3.1 Tobin Taxes

Perhaps the simplest suggested solution is for regulators or governments to simply implement a tax on each trade to the exchange (Tobin, 1978). In theory, this reduces the attractiveness of sniping opportunities and should tighten the bid/offer spread. However, in the process of addressing sniping and the race to the exchange, it also makes investors universally worse off. Models have shown that, if you assume a dollar held by the government has less social utility than one of investor profit, the Tobin Tax decreases social welfare (Budish, Che, Kojima, & Milgrom, 2013). Why penalise traditional investors for a problem that they didn't cause? Perhaps a more targeted approach, taxing speed as opposed to trades would be more appropriate (maybe one similar to that seen in (Biais, Foucault, & Moinas, 2015); yet it has been shown that the magnitude of such a tax would need to be unreasonably large to exhibit the effect desired (Budish, Che, Kojima, & Milgrom, 2013).

A proposal for a tax in this vein was made by the European Commission (EC) back in 2011 (EUR-Lex, 2021) – however, it may be a signal of the difficulty of passing such legislation that as of 2021, no such tax has been implemented. Interestingly, the EC did imply that this proposal was aimed at limiting HFT activity – noting in their FAQs that the new tax would be most irritating for 'high-frequency traders and for fund and hedge fund managers whose business model is based on quick successions of financial transactions' (European Commission, 2013).

2.3.2 Ban High-Frequency Trading

Two common characteristics of high-frequency trading are that, first, traders tend to cancel more orders than their traditional counterparts; and second, they often have a large ratio of orders to completed transactions (Securities and Exchange Commission, 2010). To counteract this, policymakers have suggested implementing minimum rest times (where a trader cannot place an order less than a given amount of time after their last order) and placing a maximum on the order : trade ratio. Implementing such rules though would go against the suggestions of Baruch and Glosten, who found that these foundations of HFT strategy are normal whilst the market is in equilibrium and these features help to promote normal market function (Baruch & Glosten, 2013).

2.3.3 Random Message Delays

The final alternative I will discuss are random message delays (Harris, 2012). Harris suggests that each posting, cancelling, and taking instruction that an exchange receives is delayed by a random amount of time (γ) between zero and ten milliseconds upon its arrival. The theory is that these delays are sufficiently large to mitigate any difference in speed between trading firms (which is in the order of microseconds), disincentivizing firms to invest in expensive technology to achieve marginal speed gains.

Again though, there are issues with this suggestion. First, it does not fundamentally change the sniping issue. If a firm attempts to cancel a stale quote, N HFT firms place orders looking to take advantage of the stale quote, who all respond equally quickly, and the random delay is much larger than the differences in speed, then a snipe will occur with probability $\frac{N-1}{N}$ (i.e., a snipe will occur in every case except the one where the cancelling party gets the smallest random delay).

Second, firms are now incentivized to place huge numbers of orders to the exchange. If we consider the example above, then each order that a firm places essentially becomes a lottery ticket in achieving their desired outcome. This may sound familiar to readers… it is another reimagining of the prisoner's dilemma! All firms now want to flood the exchange with orders, but it would clearly be in the interests of society to redesign the market so that there was once again a stable equilibrium, with each firm only placing one of the orders it wants to be executed.

3 Objectives, Specifications and Design

3.1 Objectives

To achieve the aims detailed in the introduction, a series of objectives had to be pursued. Naturally, the first was to develop a basic orderbook, exchange and codebase, off which the rest of the simulation could be built. This central exchange should have the ability to store multiple orderbooks (representing the possibility of cross-asset transactions and trading) and should have easily modifiable central components to change the market mechanism.

Following this, agents should be built on top of the basic infrastructure to yield a dynamic agent-based model, in the style of (Guo, 2005). The model should have two groups of traders – those who take liquidity, and those who provide it (via market and limit orders respectively). These two groups can be seen as an abstraction of retail and institutional investors.

Next, it is important that the agent-based model is tested, and refined, to ensure firstly that it functions, and secondly, that qualitatively resembles the type of Brownian motion seen in real markets (Guo, 2005). One way in which this was carried out was to align the trader order sizes with those observed in real financial markets (Peterson & Sirri, 2003). This step is also important to ascertain with what frequency the model breaks down (e.g., liquidity is entirely stripped from one side of the orderbook). Due to time constraints, the simulation should be built to throw an error when such fundamental issues occur.

An additional step to be performed at this point is to build a simple UI on top of the software. This increases the ease-of-use of the software and will hopefully increase the probability of it being used in the future. If possible, this should include the possibility to alter some key parameters (such as the mechanism being used and the number of each type of trader), whilst providing a live graph of how the bid-ask prices are moving over time. Furthermore, it should include some indication of how the liquidity (in this case spread) was throughout the simulation. This objective mostly satisfies our UX aim.

Once the initial foundations have been built, data can then be generated using two of the mechanisms (FCFS and LIBRA). The liquidity when using both mechanisms should be quantitatively compared. After this data is generated, claims that the researchers make about LIBRA being a potentially superior mechanism should be analysed and studied (particularly with regards to minimal impact – this is often the condition that exchanges are most concerned about when implementing new technologies). Potential advantages and disadvantages of the mechanisms could also be examined at this point.

3.2 Requirements

The software should be organised in a way that is dynamic and adaptable, yet also simple (for ease-of-use and due to time constraints), and in a way which functionally resembles real markets.

In a market, there are three key players:

- **Traders**: these agents place and cancel orders, according to some self-defined strategy.
- **Exchange**: the exchange accepts orders from the traders and matches orders together (according to the matching mechanism) to provide utility. Typically, it charges some fee for this service.
- **Dealing/Clearing House:** the third party processes the account values for each transaction, subtracting money from a traders account and passing it to another's when orders are matched together.

To perfectly mirror reality, each of these players should have their own class. However, for simplicity, the clearing house should be merged into the exchange class, and instead replaced by a simple function to update account values based on previous transactions occurring. To enable the exchange to check what has happened previously, a list of the transactions that have occurred is to be stored in the exchange as a dictionary of lists, each list representing a tick. Inside of these lists, there will be Matched Orders objects, which contain the crucial information about the transaction which has occurred.

Within the exchange, there will be a list of orderbooks – one for each asset. For now, most of the functionality (including improving efficiency) of these orderbooks working alongside each other remains outside the scope of the project (it is enough for there to simply be multiple orderbooks) – however, this is recognised as an interesting future avenue of development. In the real world, traders will often place 'pairs' or 'relative value' trades, where a bet is made on two prices moving towards or away from one another, whilst mitigating the overall market risk. The orderbook should have the ability to store orders, and quickly be able to ascertain the best bid/ask prices. Within the orderbook, there will be a flexible matching mechanism component, defined as its own object. Orders should be pushed into this matcher before being returned into the orderbook, where they can be matched. This approach has one notable shortcoming – if the auction is fundamentally redesigned (e.g., to a first-price auction), it would be challenging to design an appropriate matcher object to represent the new matching method. However, this approach does allow us to build both LIBRA and FCFS orderbooks, which meets the scope of the project. If the changeable component was changed to be the whole orderbook, more radical redesigns would be possible – however this would come with increasing complexity in ensuring all the objects work with the exchange.

The traders should be able to interact with the exchange (which then interacts with the appropriate order book), cancelling and placing orders as they see fit – just as a trader would interact with a broker in the real world. Although my model more closely resembles Direct Market Access ('DMA'), it suffices for our simulation – introducing a level of broker-dealers would make the simulation significantly more complicated, without providing a notable increase in the strength of our results (perhaps this would be an interesting addition to the project in the future?).

Further, it is important for our software to have a user interface (UI) that will allow for simple modifications to be made to the simulation variables (such as number of traders and number of steps) which should (in theory at least!) dramatically improve the user experience (UX). This UI should also have some method of monitoring the simulation – perhaps a graph of the best prices at each step. Although building an in-depth multi-layered UI is a challenging task (and beyond the scope of the project) – it is noted that 'a meaningful user experience allows you to define customer journeys… that are most conducive to business success' (Rocket55, 2021). As such, this will hopefully address our objective to have the software continue to be used in the future.

3.3 Design

In this project, the primary aim is to expand the understanding of the LIBRA mechanism and determine whether it materially changes the liquidity in a market. To make a quantitative determination on whether the two mechanisms (FCFS and LIBRA) are materially different, I will conduct a t-test on the mean of the spread (i.e., the best ask minus the best bid) across a group of simulations, with the null-hypothesis being that LIBRA is not materially different. Note that the spread is often seen to be the de-facto proxy measure for the liquidity in a market (Ganti & Scott, 2021), and by using a relatively simple metric, we can use a leaner codebase, which is easier to write and understand.

However, it is recognized that if we were to conduct a more rigorous investigation of liquidity, then the strength of our findings could be improved (indeed, this could be an interesting future direction for the project). Some work has previously been done in establishing and comparing more formal ways of defining liquidity (Ramos & Righi, 2018).

The second aim is more qualitative – to try to ensure that the software remains in use. To achieve this, I will design a user interface using the common Python3 package 'tkinter' – which comes bundled when you install Python (Python, 2021). This interface implementation is often criticized for not being the most glamorous when compared to competitors (Ramalho, 2015); however, it is easy to run and use (important given my prior lack of UI programming experience) and is also something that every user of Python3 will have access to – eliminating one potential barrier to access and helping to meet our goal of having the software used in the future.

4 METHODOLOGY AND IMPLEMENTATION

4.1 Methodology

To run the necessary simulations and obtain values for the average spread in each simulation, it will be necessary to utilise the agent-based simulation of the double auction market. To build this, I intend on drawing on the work of Guo (Guo, 2005). Within this thesis, he outlines how he built a network-based agent-based modelling platform to analyse properties of the market microstructure, such as the evolution of the market mid-price. He also goes on to do a more theoretical study of the bid-ask spread, in a similar vein to this project – but approaches the topic from a more theoretical angle, providing a comparison to a stochastic differential equation (of the form $dS = \mu \cdot dt + \sigma \cdot dW$, where S is the spread, μ a drift function depending on t, σ a time dependent volatility factor, and W a Wiener process (Durrett, 2019)).

In this work, Guo also highlights several models to generate the necessary order flow. I will briefly discuss each of these below, but for brevity, extensive discussion is not provided.

4.1.1 Chiarelli-Iori Model

This model (Chiarella & Iori, 2002) establishes a double auction marketplace using a set of heterogeneous agents, who each can post market and limit orders based on a set of predefined rules.

Once the simulation begins, an agent P_i is picked randomly with probability λ. It is assumed that the agent knows the fundamental value of the asset (which is constant through the simulation). When an agent is chosen, they compute an expectation of the spot return between the current time t and some future time $t + \tau$, based on the fundamental value and patterns that have been seen in the price evolution of the asset (representing an abstraction of technical analysis (Hayes & Brock, 2021)). Then the agent places an order in the market based on whether they feel the asset is over- or under-valued.

Although this model has strengths and resembles some aspects of my intended method (it has one asset, a relatively simple set of traders and represents the co-interaction of technical and fundamental analysis), it also has one notable shortcoming in assuming that the asset has some underlying 'true' fundamental value. This has repeatedly been shown to not be a realistic assumption in modern securities markets – see (Summers, 1986).

4.1.2 Das Model

Das (Das, 2003) proposes an alternative model to Chiarelli-Iori. He focuses on the topic of informational asymmetry, creating a market with a central market-maker and several traders, who are only able to interface with the central market-maker. This resembles the traditional NYSE dealer market[8].

In the simulation, the market maker sets bid and ask prices each tick, which the agents can transact against. The market maker is told the initial value of the asset, then updates their prices based on the amount of buying and selling done by the agents. Agents are split into two camps: informed and uninformed. Informed traders are given information about the evolving fundamental value, which changes according to a jump-point process. In the most basic form of the model, no consideration is given to the inventory held by the market-maker, but later in the paper this is added as a pricing factor (more closely resembling how market makers operate in the real world (Rill, 2015)).

Again, this model has advantages – it uses a set of informed and uninformed participants (resembling our division of retail and institutional traders into liquidity makers and takers). However, the focus on one central party determining the price is not ideal for our purposes: first, because this style of market is being replaced by ECNs, and second, in recent years there has been a growing focus on decentralized finance (DeFi), in both academia and the public eye (Koffman, 2020).

[8] Note that even more traditional exchanges, such as NYSE, now offer alternative electronic communication networks (ECNs) which avoid the need for a central market maker. NYSE's ECN is called NYSE Arca (New York Stock Exchange, 2021).

4.1.3 SFGK

The model that Guo uses is an adapted form of the one proposed by Smith, Farmer, Gillemot and Krishnamurthy (Smith, Farmer, Gillemot, & Krishnamurthy, 2003). In its original form, this model is more of a statistical one than an agent-based one, however, in Guo's work, this is adapted to be agent-based. The most compelling feature of SFGK is its ability to model extremely complex market phenomena whilst only using zero-intelligence agents who act randomly. Indeed, this echoes the seminal remarks of Gode and Sunder, who famously expressed that "Allocative efficiency of a double auction market derives largely from its structure, independent of traders' motivation, intelligence or learning. Adam Smith's invisible hand may be more powerful than some may have thought; it can generate aggregate rationality not only from individual rationality but also individual irrationality" (Gode & Sunder, 1993).

Within the original model, order flows (both limit and market) are modelled as Poisson processes. Market orders arrive in chunks of σ shares, with an arrival rate of μ shares per unit time. Limit orders also arrive in chunks of σ shares per unit time but have an arrival rate of α shares per unit time. These definitions mean that the corresponding buy/sell flows for market and limit orders have arrival rates of $\frac{\mu}{2}$ per unit time and $\frac{\alpha}{2}$ per unit price per unit time, respectively. As limit orders must also have a defined price, the log of this price is chosen from the uniform distribution $\text{Unif}(-\infty, a(t))$[9] (and $\text{Unif}(b(t), \infty)$) for buy (and sell) orders (where $a(t)$ represents the log of the best available ask price). Log values are used to ensure that price always remains positive, a common assumption for equity markets (which is due to shareholders having limited liability (Hartman, 2019)).

[9] Note that formally, according to Kolmogorov's axioms (Kolmogorov, 1933), the Uniform distribution is not definable over an infinite interval. Within the model, it is suitable to replace ∞ with *L* large – and maintain the logic of an order randomly picking a log price greater than (or less than) the best ask (bid).

The model also has orders expire according to a Poisson process with average decay rate η per unit time – computationally this is helpful in limiting memory usage throughout the simulation. The relative simplicity of the model also helps it to be analytically tractable, so that a series of claims can be made about its embedded market features – for a discussion of these see (Smith, Farmer, Gillemot, & Krishnamurthy, 2003).

In his thesis, Guo suggests having $\sigma = 1$, ie. orders always arrive separately and he also introduces the same limiting cap on the uniform distribution I discussed above. In his adaptation, all 50 traders are homogeneous and are allowed to generate both limit and market orders on both sides of the book.

4.1.4 Our Model

Within our model, parts of various models are used, but it is mainly an adapted form of the one Guo uses in his simulation. There are two stages – initiation and then a main loop is run. Initially, the orderbook and traders need to be created, and the orderbook primed appropriately – to define a best bid and ask price. The traders will be in two camps, as previously mentioned – those placing liquidity, the 'makers' and those taking liquidity, the 'retail'. The number of each in the software is variable depending on the parameters of a given simulation run.

After the orderbook is generated, there will be a buy and sell order created (priced at 9.5 and 12.5 respectively) and instantly added to the orderbook (see step 1 in Figure 6) – disregarding any matching mechanism. After this, the 'initiation phase' of the simulation begins (step 2 in Figure 6). Traders are filtered so that only the market makers can act – who are allowed to add liquidity to the book via their order generation process. After a set number of 'ticks' in the simulation, all traders are then able to act – both types can now place market/limit orders (step 3 in Figure 6). Note that in every tick, the order of the traders as they come to the market is randomised – ensuring no trader has an advantage in accessing the market before

other participants.[10] In real world terms, this is an abstraction of all traders being equal in ability and equidistant from the exchange.

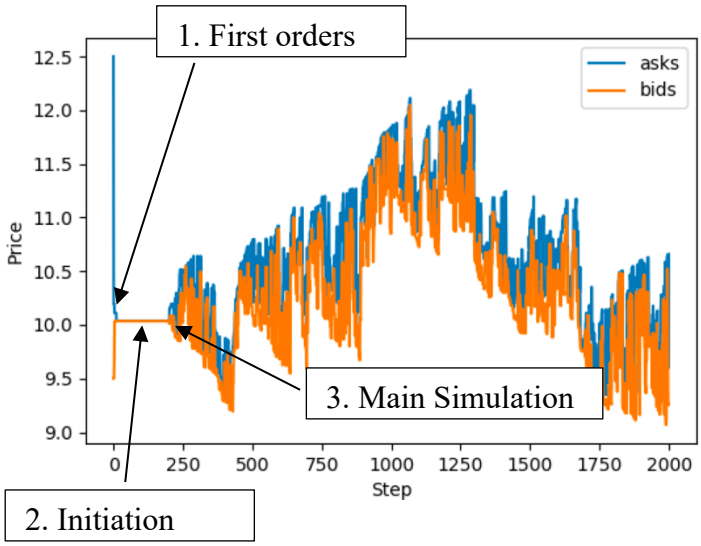

Figure 6: Phases of simulation, shown on software output graph.

When a trader is selected by the simulation, they first cancel a set percentage ξ of their outstanding orders. After this, the trader 'calls' the exchange and has the ability to place a trade. Individual traders generate orders according to uniform distributions as seen in SFGK (Smith, Farmer, Gillemot, & Krishnamurthy, 2003). To decide whether to trade, first a number is randomly drawn from the interval $(0,1)$. If the number selected is less than the

[10] One interesting future direction of the project could involve changing this random shuffling mechanism to one where certain traders more often arrive earlier to the exchange – perhaps reflecting greater investments in speed technologies or colocation. Perhaps this could take the form of some type of weighted permutation function as seen in (Helmbold & Warmuth, 2009).

probability of placing a trade, then the trader decides whether it should be a buy or sell order with a similar process. Once the type of order is determined, the price of an order is determined as in SFGK – with the trader drawing the log price from the appropriate uniform distribution Unif$(b(t), L)$. However, I discretize the price into ticks (as is seen in real exchanges), and after the random price is found, this is rounded to the nearest tick.

Following this, the volume of the submitted order is decided. This is done by drawing another random number from the unit interval – to determine whether the trade should be of small, medium, or large volume. Once banded, the volume of the trade is drawn uniformly on the interval containing the limits of each of the bands. This process is based on the work of Peterson and Sirri (Peterson & Sirri, 2003), who collated data on the sizes of limit orders that were placed on NYSE.

4.2 Implementation

4.2.1 Main Simulation Logic

As mentioned below, the main simulation logic ('game loop') is defined within the **runSimulation()** function (this can be seen in Appendix 1 – 'main.py') – which is called when the button is pressed in the UI. An instance of the exchange is created – then the traders are created using a list comprehension. There is then a loop, within which traders are called given the chance to act (first in the initiation period, then the main simulation) as defined earlier in this chapter.

4.2.2 Trader Class(es)

Traders in the simulation are represented using the **Trader()** class (seen in Appendix 2 – 'traders.py'), below which there are two sub-classes – **MarketMaker()** and **RetailTrader()**. The header class contains functions all traders need to have – such as calling the exchange – whereas the sub-classes contain the individual logic for the trader to run, and how it will decide whether to submit an order. One of the strengths of the object-oriented nature of our software is that it would be relatively easy to configure further trader classes, such as Momentum or Value traders.

Within each trader's run (**.run()**) function, it is first decided whether to cancel orders, and then it calls the exchange (containing an order if it wishes to place one). As previously mentioned, in the current version, the trader i cancels n orders according to the discrete uniform distribution Unif$\left(0, N_{O,i}\right)$ where $N_{O,i}$ represents the number of outstanding orders that a trader has placed. To track the number of 'live' orders, the trader maintains a buffer of orders. This is implemented as a list, which contains order objects representing those held at the exchange. When a trader places a new order, said order is added to the buffer. Additionally, whenever **.run()** is called, all orders in the buffer are checked with the exchange to check whether they still exist (and what portion of them have been matched).

4.2.3 Orderbook

So that it can store orders according to the specifications, the orderbook will have three notable parts (see Appendix 3 – 'orderbook.py'): a dictionary of lists - containing the order objects at each price level, a set containing all the ask prices and a set containing all the bid prices. These will be updated any time an order is to be added after it has made it through the appropriate matching object. By using sets, it is easy to call a $\max()$ or $\min()$ function in $O(n)$ time (Tripathi, 2018) – efficient enough when we do not require live access to the exchange during runtime. Whenever the orderbook is initiated, it also creates an instance of the appropriate matching object (FCFS or LIBRA). This means that the orderbook is able to forward orders into the matcher whenever a new order is added to the orderbook. It is also important to note that any time the orderbook is called, it checks with the matching object to see whether any orders need to be added back into the orderbook.

4.2.4 LIBRA matcher

Originally, Mavroudis and Melton describe the cancelling of orders as being a type of order that bypasses the buffer system. However, cancellation is arranged via a different system (see Appendix 4 – 'LIBRAMatcher.py') in the software – a method to cancel an order is called, as opposed to submitting an order with the cancel type. Although this is a slight modification to the original algorithm, it should not fundamentally alter the results. In both cases, the request to cancel the order is acted upon instantly – the only difference being that the exchange knows when a trader wants to cancel an order at the point of submission, as opposed to when it processes the order.

4.2.5 User Interface

The UI was implemented using the Tkinter package, as previously mentioned in the report. Using this package, it is possible to construct relatively complicated interface designs. However, due to a lack of any experience in the area, I decided to keep things relatively simple. Within the *main.py* file, a window is initiated and then populated with entry fields, labels and the appropriate button using simple Tkinter objects and methods. Following this, the graph that can be seen is created using the matplotlib library, and its sub package FigureCanvasTkAgg – a figure (and subplot) is created, then this is turned into a 'canvas', before being added to the window.

Within Tkinter, there are three built in layout-managers: **.grid(), .pack()** and **.place()** (ZetCode, 2020). Each of these have pros and cons, but for the (geometrically) simple layout in our software, **.grid()** was used (see Appendix 1 – 'main.py'). This allows for items to be set out in rows and columns, with items also being allowed to span columns/rows where necessary – such as the graph on the right of our window.

When the 'Run Simulation!' button is clicked, it triggers the **runSimulation()** function. This initiates the appropriate objects, and then runs the simulation, finally updating the mean spread label and graph subplot to show the results. Although the UI is simple, I believe it is effective in meeting our aims. It enables for simulations to be easily run, and parameters altered, whilst also providing an interesting visual stimulus (in the form of the price graph).

Given more time working on the project, it would have been interesting to develop this UI further – perhaps using a system of front-end classes for different windows that could be accessed, showing different views of the simulation, or allowing for more complex parameter adjustment (e.g., the probability that a trader places an order). In theory, it could even be possible to have the user interact with the simulation themselves, placing buy/sell orders into the market – whilst also monitoring their P/L (profit and loss (BenchMark, 2021)). This is noted as one potential future direction of the work.

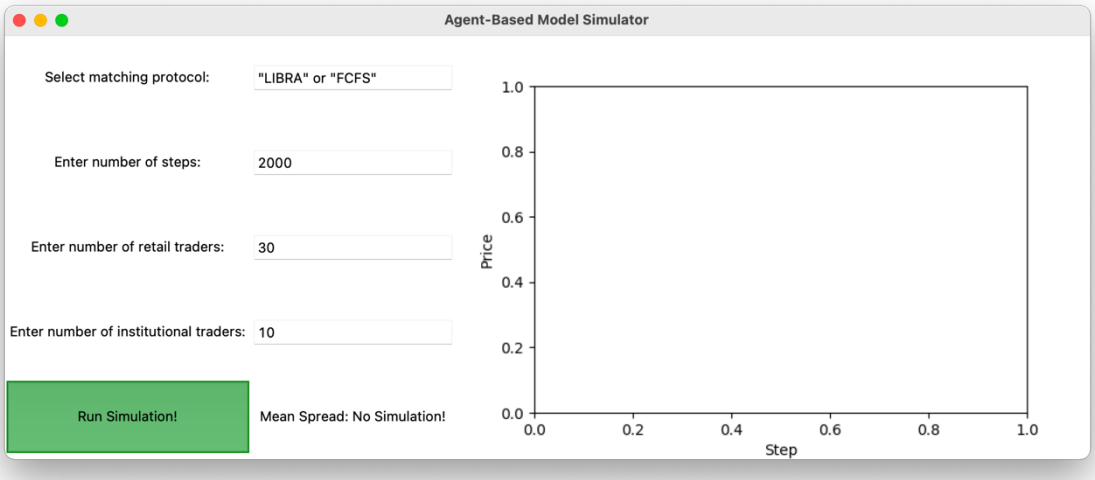

Figure 7: User Interface which appears before a simulation run.

Figure 8: User Interface after a simulation has been performed.

5 Results, Analysis and Evaluation

Using the software, groups of simulation were run using both LIBRA and FCFS. Initially, sets of simulations were run to determine stable numbers of both retail/institutional traders. It was found that by using twenty retail agents, and ten institutional agents, the simulation remained stable across all twenty initial test runs. By changing the ratio of these agents, you can manipulate the ratio of liquidity takers to givers – by increasing the number of retail traders, it was found that (qualitatively) the spread tended to increase, and the probability of the simulation ending early due to one side of the book getting 'stripped'[11] of all liquidity also increased.

For the main results, twenty simulations were performed using both LIBRA and FCFS. The mean of the spread in each scenario can be seen in the table below (Table 1) – along with the average across each group of simulations.

LIBRA	FCFS
0.27546	0.2947

[11] This term refers to a liquidity crisis whereby all limit orders have been exercised on one side of the orderbook, leaving no liquidity for price takers. In the software, this results in the best bid/ask no longer being defined – which causes the simulation to crash. Perhaps an interesting future line of study would be to investigate what mechanisms can be used when there are no limit orders on one side of the book – should the exchange insert orders itself in this scenario?

0.28223	0.31034
0.29265	0.29273
0.30489	0.28821
0.28187	0.29241
0.28049	0.30011
0.28983	0.28424
0.2786	0.27492
0.30975	0.30099
0.27095	0.28812
0.28567	0.26572
0.27802	0.27832
0.28056	0.3095
0.28297	0.28435
0.27748	0.30193
0.27463	0.28144
0.29004	0.28725
0.29695	0.29191
0.28102	0.28734
0.29638	0.29027
0.285522	0.29024

Table 1: The mean spread in each simulation.

As previously mentioned, we will use a hypothesis test (independent t-test) to determine whether the spread is materially affected by changing the matching mechanism from FCFS to LIBRA. The null hypothesis, h_0 is that $\mu_{\text{FCFS}} = \mu_{\text{LIBRA}}$. The alternative hypothesis, h_1 is that $\mu_{\text{FCFS}} \neq \mu_{\text{LIBRA}}$ – as a result our test is two tailed.

The 20 simulations using LIBRA (M = 0.285522, SD = 0.01027) compared to the 20 simulations that we used as a control (FCFS) (M = 0.29024, SD = 0.01103), did not demonstrate a difference in the mean spread (t(38) = 1.40047, p = .16948), when evaluated at a significance level of 0.05. Hence, we can conclude that, within our model, LIBRA does not materially affect the spread when compared to FCFS. This suggests that exchanges could stand to benefit from switching to this mechanism in the future (or should at least conduct further tests). However, throughout this research we have not considered the frictions in moving to a new matching technology, designing the necessary algorithms, and acquiring the necessary software. Whether these costs outweigh the benefits that LIBRA provides (in preventing the HFT arms race) would need to be studied further to provide a conclusive argument. To do this, it is likely that surveys and data would need to be collected to determine the utility (both social and monetary) that LIBRA would yield.

Another interesting feature of our results is the wide variety of simulation outcomes that can occur, despite using very simple (zero-intelligence) trading methodology. As can be seen in the figures below (Figure 9 & Figure 10) – it is possible for both the price to be constrained to a range or diverge outside the tight band. It is also worth noting that the initial primer orders (at 9.5 & 12.5) have a very widespread compared to the rest of the simulation. This is intentional – it means that the initial orders have little effect on constraining the spread once

the simulation begins, as during the initiation phase, this spread serves as an initial upper bound (the market makers place limit orders which are not allowed to cross the book).

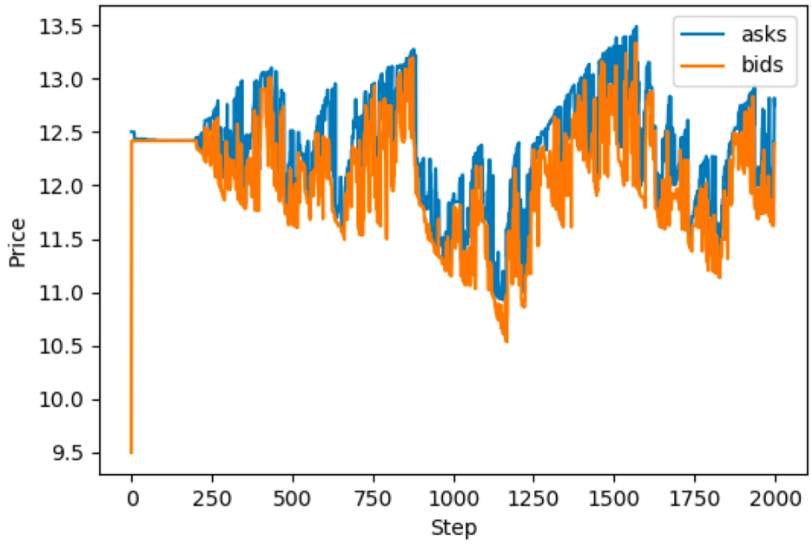

Figure 9: Output graph from software showing a relatively bounded path.

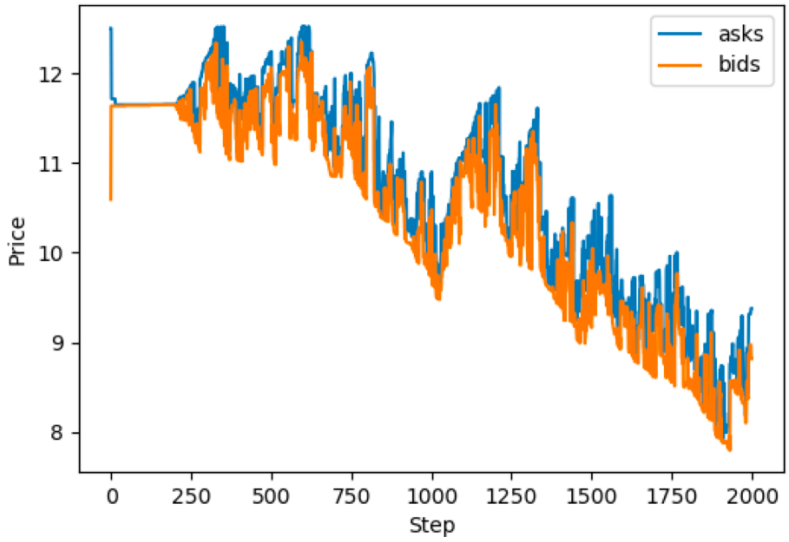

Figure 10: Output graph from software showing a path with more variance.

6 Legal, Social, Ethical and Professional Issues

Through this project I have tried to follow the Code of Conduct and Code of Good Practice, published by the British Computer Society (BCS) (British Computer Society, 2021). These codes stand on four pillars: 'You make IT for everyone', 'Show what you know, learn what you don't', 'Respect the organisation or individual you work for' and 'Keep IT real. Keep IT professional. Pass IT on.' I address each of these in turn below.

6.1 You make IT for everyone

This is closely related to one of the aims of the project – promoting use of the software and removing entry barriers where possible to preserve its use moving forwards. The accessibility of the project is also improved via the addition of the simple UI, which allows those from a non-technical background to get a grasp on the project, what it is about, and what it can do.

This principle also includes respecting the rights of third parties, which I have tried to accomplish by citing research, papers, and articles appropriately in the APA style throughout this project. I believe that I have also only used freely available python packages, which have no licensing requirements for our non-commercial purposes.

6.2 Show what you know, learn what you don't

This important tenet has been addressed by limiting the scope of the project to what is accomplishable within the set timeline. Sometimes, this has been due to a lack of previous experience on my part (e.g., with the UI), and at other times because the task would have resulted in the project over-running (e.g., coding more traders). However, I have also repeatedly highlighted directions the project could go in given more resources.

I have also found that this project has been a fantastic opportunity to learn and develop, both for technical skills (such as using Tkinter, or learning more about matching algorithms), as well as softer skills (such as discussions with supervisors).

6.3 Respect the organisation or individual you work for

I believe that this point is perhaps less relevant to my project, although I have tried to respect all individuals I have encountered during the project and have also respected the work of others by citing it where appropriate. Furthermore, I have mentioned some of the limitations of the work throughout this report, and how these could be mitigated given more time.

6.4 Keep IT real. Keep IT professional. Pass IT on.

This clause is something that I have striven to achieve – to uphold the standards of the IT and computer science communities. With regards to the last part ‘pass IT on’ – as I have mentioned above, this is very applicable to this project. I would also be more than happy to discuss this report further with any future reader who wishes to reach out.

7 CONCLUSION

7.1 Discussion

In this report, I have described some of the prevailing issues with market mechanisms, and then gone on to discuss the pros and cons of some of the suggested solutions in the literature. To test one of these alternatives against the status quo, a flexible simulation platform was developed, which would allow for further algorithms to be written as matching objects. When evaluating the results generated by the model, it was found that LIBRA does not materially change the liquidity of a marketplace, and hence would be suitable for use in major exchanges. This is in line with Mavroudis and Melton's suggestion that any replacement mechanism should have minimal market impact (Mavroudis & Melton, 2019).

The agent-based simulation platform was useful and novel. The software includes a helpful user interface to increase the accessibility and usefulness of the project moving forwards.

To conclude, LIBRA presents a unique opportunity to prevent the 'race to the exchange', increasing social welfare for all market participants. However, exchanges (and their shareholders) are forever fixated on maintaining their competitive advantage(s). Based on the findings of this project, these exchanges can be reassured that their liquidity will not be significantly affected by changing to a new mechanism.

7.2 Future Work

There are several future directions in which this work could progress, some of which I have highlighted throughout this work. The first notable one would be to implement the FBA into the simulation and examine how (if at all) it changes the market dynamics when compared to LIBRA & FCFS.

Next, the strength of the finding that LIBRA does not materially change the market liquidity would be improved if a more robust method of testing liquidity was used – as opposed to running a hypothesis test on the mean of the spread. As I previously mentioned, some work has been done in defining more rigorous measurements for liquidity (Ramos & Righi, 2018).

Notably, one underdeveloped area of the software is the UI. This could be made more complex, having multiple windows for varying more complicated inputs and observing different views of the market.

Finally, perhaps the most interesting area would be to develop more intelligent traders – those using known strategies such as momentum or value, or alternatively, novel ones not yet seen. It could be interesting to have students develop their own traders, and then have them compete in the marketplace, using the account value tracker to see who does better. This development could also include implementing cross-asset traders.

8 BIBLIOGRAPHY

Aldridge, I. (2013). *High-Frequency Trading: A Practical Guide to Algorithmic Strategies and Trading Systems.* Wiley Trading.

Amihud, Y., & Mendelson, H. (1988). Liquidity and Asset Prices: Financial Management Implications. *Financial Management*, 5-15.

Anderson, B. (2021, August 2). *Instant Replay: Conley Shares Wolf Of Wall Street Tribute In Jazz Return*. Retrieved from KSL Sports: https://kslsports.com/465018/instant-replay-conley-shares-wolf-of-wall-street-tribute-in-jazz-return/

Aquilina, M., Budish, E., & O'Neill, P. (2021). Quantifying the high-frequency trading "arms race". *BIS Working Papers*.

Bae, S. (2019). Big-O Notation: An Introduction to Understanding and Implementing Core Data Structure and Algorithm Fundamentals. In *JavaScript Data Structures and Algorithms* (pp. 1-11).

Baruch, S., & Glosten, L. R. (2013). Fleeting Orders. *Columbia Business School Research Paper 13-43*, 1-34.

BenchMark. (2021, August 30). *P/L Calculations for Stocks*. Retrieved from BenchMark: https://benchmarkfx.fr/pics/editor/PDFFR/PnL_Calculations_for_Stocks_ESP.pdf

Bernholz, P. (2003). *Monetary Regimes and Inflation: History, Economic and Political Relationships.* Edward Elgar Publishing.

Biais, B., Foucault, T., & Moinas, S. (2015). Equilibrium fast trading. *Journal of Financial Economics*, 292-313.

Breckenfelder, J. (2020, December 15). *How does competition among high-frequency traders affect market liquidity?* Retrieved from European Central Bank: https://www.ecb.europa.eu/pub/economic-research/resbull/2020/html/ecb.rb201215~210477c6b0.en.html

British Computer Society. (2021, August 28). *BCS Code of Conduct*. Retrieved from BCS: https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/

Brogaard, J. (2010). High Frequency Trading and Volatility. *SSRN Electronic Journal*.

Brown, A. (2015, August 11). *HFT 'speed race to zero' is almost over, says Norges.* Retrieved from IR magazine: https://www.irmagazine.com/reporting/hft-speed-race-zero-almost-over-says-norges

Budish, E., Che, Y.-K., Kojima, F., & Milgrom, P. (2013). Designing Random Allocation Mechanisms: Theory and Applications. *American Economic Review*, 585-623.

Budish, E., Lee, R. S., & Shim, J. (2019). A Theory of Stock Exchange Competition and Innovation: Will the Market Fix the Market? *BFI Working Paper*.

Caivanoo, V. (2015). The Impact of High-Frequency Trading on Volatility. Evidence from the Italian Market . *CONSOB Working Papers no. 80*.

Chen, J., & Scott, G. (2021, May 27). *E-Mini*. Retrieved from Investopedia: https://www.investopedia.com/terms/e/emini.asp

Chiarella, C., & Iori, G. (2002). A simulation analysis of the microstructure of double auction markets. *Quantitative Finance*, 346-353.

Colt. (2011, January 10). *Colt launches new proximity hosting service in partnership with London Stock Exchange*. Retrieved from Colt: https://www.colt.net/resources/colt-launches-new-proximity-hosting-service-in-partnership-with-london-stock-exchange/

Corporate Finance Institute. (2021, August 24). *Dutch Tulip Bulb Market Bubble*. Retrieved from Corporate Finance Institute: https://corporatefinanceinstitute.com/resources/knowledge/economics/dutch-tulip-bulb-market-bubble/

Daian, P., Goldfelder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., . . . Juels, A. (2019). Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges. *arXiv*.

Das, S. (2003). *Intelligent Market-Making in Artificial Financial Markets.* Boston: Massachussetts Institute of Technology.

Duhigg, C. (2009, July 23). *Stock Traders Find Speed Pays, in Milliseconds*. Retrieved from New York Times: https://www.nytimes.com/2009/07/24/business/24trading.html

Durrett, R. (2019). Brownian Motion. In R. Durrett, *Probability: Theory and Examples.*

Etheridge, J. (2021, March 16). *The history of investing: What the past can teach us about market cycles*. Retrieved from Focused Financial: https://focusedfinancial.co.uk/the-history-of-investing-what-the-past-can-teach-us-about-market-cycles/

EUR-Lex. (2021, August 25). *Council Directive implementing enhanced cooperation in the area of financial transaction tax - Policy debate/progress report ST 10097 2019 INIT*. Retrieved from EUR-Lex: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=consil:ST_10097_2019_INIT

European Commission. (2013). *FTT: NON-TECHNICAL ANSWERS TO SOME QUESTIONS ON CORE FEATURES AND POTENTIAL EFFECTS.* Brussels: European Commission.

Ganti, A., & Scott, G. (2021, March 18). *Bid-Ask Spread*. Retrieved from Investopedia: https://www.investopedia.com/terms/b/bid-askspread.asp

Gode, D. K., & Sunder, S. (1993). Allocative Efficiency of Markets with Zero-Intelligence Traders: Market as a Partial Substitute for Individual Rationality. *Journal of Political Economy*, 119-137.

Goldman, H. (2012, November 27). *What is driving co-location and proximity hosting in Asia?* Retrieved from GlobalTrading: https://www.fixglobal.com/what-is-driving-co-location-and-proximity-hosting-in-asia/

Gomber, P., Arndt, B., Lutat, M., & Uhle, T. (2011). *High-Frequency Trading.* Frankfurt Am Main: Goethe Universitat.

Guo, T. X. (2005). *An Agent Based Simulation of Double Auction Markets.* Toronto: Toronto University. Retrieved from Toronto University CS.

Harris, L. (2012, December 27). *Stop the high-frequency trader arms race*. Retrieved from The Financial Times: https://www.ft.com/content/618c60de-4b80-11e2-88b5-00144feab49a

Harris, L. (2013). What to do about high-frequency trading. *Financial Analysis Journal*.

Hartman, D. (2019, February 19). *Can Stock Value be Negative?* Retrieved from Zacks: https://finance.zacks.com/can-stock-value-negative-9119.html

Hayes, A. (2021, July 19). *Call Auction*. Retrieved from Investopedia: https://www.investopedia.com/terms/c/call-auction.asp

Hayes, A., & Brock, T. (2021, August 10). *Technical Analysis*. Retrieved from Investopedia: https://www.investopedia.com/terms/t/technicalanalysis.asp

Helmbold, D., & Warmuth, M. K. (2009). Learning Permutations with Exponential Weights. *Journal of Machine Learning Research*, 1705-1736.

Hinterleitner, G., Leopold-Wildburger, U., Mestel, R., & Palan, S. (2015). A Good Beginning Makes a Good Market: The Effect of Different Market Opening Structures on Market Quality. *Probability and Statistics with Applications in Finance and Economics*.

Holmes, F. (2015, June 26). *There's a Huge Disparity Between How Regulators Deal with Gold and Stock Market Manipulation*. Retrieved from US Global Investors: https://www.usfunds.com/investor-library/frank-talk-a-ceo-blog-by-frank-holmes/theree28099s-a-huge-disparity-between-how-regulators-deal-with-gold-and-stock-market-manipulation/#.YTBCdy0RpAY

ice. (2021, August 24). *Colocation & Hosting*. Retrieved from ice: https://www.theice.com/market-data/connectivity-and-feeds/colocation-and-hosting

Klocking, J.-U., Maihofer, C., & Rothermel, K. (2001). Reducing multicast inter-receiver delay jitter - a server based approach. *International Conference on Networking* (pp. 498-507). Springer.

Koffman, T. (2020, August 31). *DeFi: The Hot New Crypto Trend Of 2020*. Retrieved from Forbes: https://www.forbes.com/sites/tatianakoffman/2020/08/31/defi-the-hot-new-crypto-trend-of-2020/?sh=12096a6e5bce

Kolmogorov, A. N. (1933). *Foundations of the Theory of Probability.* New York: Chelsea Publishing Company.

Lacasse, M. (2012, February 9). *Nano is Nasdaq's New Microsecond*. Retrieved from Lightspeed: https://www.lightspeed.com/active-trading-blog/nano-is-nasdaqs-new-microsecond/

Lewis, M. (2014). *Flash Boys: A Wall Street Revolt.* W. W. Norton & Company.

Li, J., Luo, S., & Zhou, G. (2021). Call auction, continuous trading and closing price formation. *Quantitative Finance*, 1037-1065.

Lioudis, N., & Silberstein, S. (2021, May 24). *Commodities Trading: An Overview*. Retrieved from Investopedia: https://www.investopedia.com/investing/commodities-trading-overview/

Mavroudis, V., & Melton, H. (2019). Libra: Fair Order-Matching for Electronic Financial Exchanges. *Proceedings of the 1st ACM Conference on Advances in Financial Technologies.* Zurich: Association for Computing Machinery.

National Museum of Australia. (2021, August 24). *First stock exchange*. Retrieved from National Museum Australia: https://www.nma.gov.au/defining-moments/resources/first-stock-exchange

New York Stock Exchange. (2021, August 28). *NYSE Arca Equities*. Retrieved from NYSE: https://www.nyse.com/markets/nyse-arca

Osipovich, A. (2017, December 19). *Court Rules for Lawsuit Against Stock Exchanges to Proceed*. Retrieved from Wall Street Journal: https://www.wsj.com/articles/court-rules-for-lawsuit-against-stock-exchanges-to-proceed-1513716880

Peterson, M., & Sirri, E. (2002). Order Submission Strategy and the Curious. *Journal of Financial and Quantitiative Analysis*.

Peterson, M., & Sirri, E. (2003). Order Preferencing and Market Quality on U.S. Equity Exchanges. *Review of Financial Studies*, 385-415.

Phillips, M. (2011, March 28). *Should High-Frequency Trading Be Banned? One Nobel Winner Thinks So*. Retrieved from Freaknomics: https://freakonomics.com/2011/03/28/should-high-frequency-trading-be-banned-one-nobel-winner-thinks-so/

Poirier, I. (2012). High-Frequency Trading and the Flash Crash: Structural Weaknesses in the Securities Markets and Proposed Regulatory Responses. *Hastings Business Law Journal*.

Polianskaya, A. (2018, March 15). *Humans may have been trading with each other as far back as 300,000 years*. Retrieved from i news: https://inews.co.uk/news/science/early-humans-trading-300000-years-135655

Python. (2021, August 26). *tkinter — Python interface to Tcl/Tk*. Retrieved from Python: tkinter — Python interface to Tcl/Tk

QuantHouse. (2021, August 24). *Proximity Hosting Services*. Retrieved from QuantHouse: https://www.quanthouse.com/colocation_proximity_hosting_services.html

Quintenz, B. (2019, May 15). *Statement of Commissioner Brian D. Quintenz on the Certification of ICE Futures U.S., Inc. Submission No. 19-119.* Retrieved from Commodity Futures Trading Commission: https://www.cftc.gov/PressRoom/SpeechesTestimony/quintenzstatement051519

Ramalho, L. (2015). *Fluent Python: Clear, Concide and Effective Programming.* O'Reilly Media.

Ramos, H. P., & Righi, M. B. (2018). Semi-deviation-based measures of liquidity. *SSRN Electronic Journal*.

Reuters. (2021, May 5). *CME to permanently close most of its physical trading pits*. Retrieved from CNBC: https://www.cnbc.com/2021/05/05/cme-to-permanently-close-most-of-its-physical-trading-pits-.html

Rill, E. (2015). *The Role of Market-Maker/Dealer Inventories in the Price Formation Process.* GRIN.

Rocket55. (2021, August 23). *The Importance of User Experience*. Retrieved from Rocket55: https://www.rocket55.com/lab-note/the-importance-of-user-experience/

Roth, A. E. (2002). THE ECONOMIST AS ENGINEER: GAME THEORY, EXPERIMENTATION, AND COMPUTATION AS TOOLS FOR DESIGN ECONOMICS. *Econometrica*, 1341-1378.

Securities and Exchange Board of India. (2019, April 30). *Order in the matter of NSE Colocation [SAT Appeal No.: 276/2019]*. Retrieved from Securities and Exchange Board of India: https://www.sebi.gov.in/enforcement/orders/apr-2019/order-in-the-matter-of-nse-colocation_42880.html

Securities and Exchange Commission. (2010). *FINDINGS REGARDING THE MARKET EVENTS OF MAY 6, 2010.* New York: CFTC & SEC.

Smith, E., Farmer, J. D., Gillemot, L., & Krishnamurthy, S. (2003). Statistical theory of the continuous double auction. *Statistical Mechanics*.

Sornette, D., & von der Becke, S. (2011). *Crashes and High Frequency Trading.* London: Government Office for Science.

Summers, L. H. (1986). Does the Stock Market Rationally Reflect Fundamental Values? *Vol. 41, No. 3, Papers and Proceedings of the Forty-Fourth Annual Meeting of the America Finance Association* (pp. 591-601). New York: Wiley.

Thompson, E. A. (2007). The Tulipmania: Fact or Artifact? *Public Choice*, 99-114.

Tobin, J. (1978). A Proposal for International Monetary Reform. *Eastern Economic Journal*, 153-159.

Tripathi, S. (2018, March 11). *Time complexity of min, max on sets*. Retrieved from stack overflow: https://stackoverflow.com/questions/49217121/time-complexity-of-min-max-on-sets

Verity, A., Lawrie, & Eleanor. (2020, January 28). *Hound of Hounslow: Who is Navinder Sarao, the 'flash crash trader'?* Retrieved from BBC News: https://www.bbc.com/news/explainers-51265169

World Federation of Exchanges. (2021, August 25). *Statistics*. Retrieved from World Federation of Exchanges: https://www.world-exchanges.org/our-work/statistics

ZetCode. (2020, July 6). *Layout management in Tkinter*. Retrieved from ZetCode: https://zetcode.com/tkinter/layout/

9 APPENDICES

9.1 main.py

```
"""Run this file."""

from traders import MarketMaker, RetailTrader
from exchange import Exchange
from order import Order
import globals as gs

import matplotlib
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib import style
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

import time
import random
import tkinter as tk
import numpy as np

def runSimulation():
    gs.NUMBER_OF_STEPS = int(sE.get())
    gs.MATCHING_PROTOCOL = mE.get()
    gs.NUMBER_OF_RETAIL = int(rE.get())
    gs.NUMER_OF_MAKERS = int(iE.get())

    sdLT = tk.StringVar()
    sdLT.set("Mean Spread: In Progress!")
    sdL = tk.Label(window, textvariable=sdLT, height=4).grid(row=4, column = 1)

    ASSETS = [gs.ASSET_CODE,'AAPL']
    TEST_EX = Exchange(ASSETS)
```

```
    # these orders can be changed if necessary - they simply allow the orderbook to have two
    # intial orders for a best bid/ask to be defined
    testOrder = Order('001',10,gs.ASSET_CODE,1,0,9.5,'firstbuy')
    testOrder1 = Order('002',20,gs.ASSET_CODE,1,1,12.5,'firstsell')

    maker1 = MarketMaker('maker', TEST_EX, 2000)
    makers = [MarketMaker('maker'+str(i), TEST_EX, 2000) for i in
range(gs.NUMBER_OF_MAKERS)]
    retail = [RetailTrader('retail'+str(i),TEST_EX, 2000) for i in
range(gs.NUMBER_OF_RETAIL)]

    #place test orders
    TEST_EX.place_order(testOrder, 0)
    TEST_EX.place_order(testOrder1, 0)

    best_asks = []
    best_bids = []
    spread = []
    traders = makers+retail

    print("starting simulation")
    # for step in range(gs.NUMBER_OF_STEPS):
    #     ani=animation.FuncAnimation(f, runStep, interval = 200, fargs=(TEST_EX, traders,
step, best_bids, best_asks, spread))

    for i in range(gs.NUMBER_OF_STEPS):
        TEST_EX.matched_orders_dict[i] = []

        random.shuffle(traders)
        for trader in traders:
            # if we are in the first 200 steps, then only allow limit orders that don't cross
            # the spread to be placed
            if i<200:
                if trader.type == "market_maker":
                    trader.run(0.25,0.5,i)
                elif trader.type == "retail":
                    pass
                else:
                    raise Exception("unknown trader type")
            else:
                if trader.type == "market_maker":
                    trader.run(0.25,0.5,i)
                elif trader.type == "retail":
                    trader.run(0.05,0.5,i)
                else:
                    raise Exception("unknown trader type")


        best_asks.append(TEST_EX.get_best_ask(gs.ASSET_CODE))
```

```
        best_bids.append(TEST_EX.get_best_bid(gs.ASSET_CODE))

        TEST_EX.revise_account_values(i)

        spread = []
        zip_obj = zip(best_asks,best_bids)
        for x,y in zip_obj:
            spread.append(x-y)

    a.clear()
    a.plot(best_asks, label="asks")
    a.plot(best_bids, label="bids")
    a.legend()
    a.set_xlabel('Step')
    a.set_ylabel('Price')
    canvas = FigureCanvasTkAgg(f, window)
    # canvas.show()
    canvas.get_tk_widget().grid(row=0,column=2, rowspan=5)

    sdLT = tk.StringVar()
    sdLT.set("Mean Spread:"+str(round(np.mean(spread),5)).rjust(15))
    sdL = tk.Label(window, textvariable=sdLT, height=4).grid(row=4, column = 1)


f = plt.Figure(figsize=(6,4), dpi=100)
a = f.add_subplot(111)
a.set_xlabel('Step')
a.set_ylabel('Price')

# Simple UI code
window = tk.Tk()
window.title("Agent-Based Model Simulator")

mLT = tk.StringVar()
mLT.set("Select matching protocol:")
mL = tk.Label(window, textvariable=mLT, height = 4).grid(row=0, column=0)

mET = tk.StringVar(None)
mE = tk.Entry(window, textvariable=mET, width = 20)
mE.insert(tk.END, '"LIBRA" or "FCFS"')
mE.grid(row=0, column=1)

sLT = tk.StringVar()
sLT.set("Enter number of steps:")
sL = tk.Label(window, textvariable=sLT, height = 4).grid(row=1, column=0)

sET = tk.StringVar(None)
sE = tk.Entry(window, textvariable=sET, width = 20)
sE.insert(tk.END, '2000')
sE.grid(row=1, column=1)
```

```
rLT = tk.StringVar()
rLT.set("Enter number of retail traders:")
rL = tk.Label(window, textvariable=rLT, height=4).grid(row=2, column=0)

rET = tk.StringVar(None)
rE = tk.Entry(window, textvariable=rET, width = 20)
rE.insert(tk.END, '30')
rE.grid(row=2,column=1)

iLT = tk.StringVar()
iLT.set("Enter number of institutional traders:")
iL = tk.Label(window, textvariable=iLT, height=4).grid(row=3, column=0)

iET = tk.StringVar(None)
iE = tk.Entry(window,textvariable=iET, width = 20)
iE.insert(tk.END, '10')
iE.grid(row=3,column=1)

button = tk.Button(
   text="Run Simulation!",
   width=25,
   height=4,
   highlightbackground="green",
   command=runSimulation
)
button.grid(row=4, column=0)

sdLT = tk.StringVar()
sdLT.set("Mean Spread: No Simulation!")
sdL = tk.Label(window, textvariable=sdLT, height=4).grid(row=4, column = 1)

canvas = FigureCanvasTkAgg(f, window)
canvas.get_tk_widget().grid(row=0,column=2, rowspan=5)

window.mainloop()
```

9.2 traders.py

```
from order import Order
import random
import math
import numpy as np
import globals as gs

class Trader():
   def __init__(self, id, exchange):
      self.id = id
      self.order_num = 0
```

```
        self.exchange = exchange
        self.order_buffer = []

    # uses the new call exchange method
    def call_exchange(self, tick, order=None):
        if order:
            self.order_buffer.append(order)
            self.exchange.call_book(tick, order)
        else:
            self.exchange.call_book(tick)

    def update_buffer(self):
        """
        here the code checks through the orders in the buffer, and sees if they were matched in
the last tick, or how they were amended
        """
        # if len(self.order_buffer) > 0:
        for order in self.order_buffer:
            # check if the order was in the matched orders from the last tick
            # print(order)
            vol_left = self.exchange.check_order_volume(order)
            order.volume = vol_left

        # clean any orders with zero volume left

        self.order_buffer = [x for x in self.order_buffer if x.volume > 0]

    def limit_volume_generator(self, max_vol):
        """
        this will determine the volume of a given order
        """
        # pick a small, medium or large order
        type = random.uniform(0,1)

        # the probabilities of each type of order are hardcoded based on paper
        if type <= gs.PROB_SMALL_LIMIT_ORDER:
            vol = random.randint(100,500)
        elif type <=
(gs.PROB_SMALL_LIMIT_ORDER+gs.PROB_MEDIUM_LIMIT_ORDER):
            vol = random.randint(501,1000)
        else:
            vol = random.randint(1000,max_vol)
        return vol

    def cancel_orders(self, cancel_prob):
        random.shuffle(self.order_buffer)
        number_to_cancel = math.floor(cancel_prob*len(self.order_buffer))
        for i in range(number_to_cancel):
            self.exchange.cancel_order(self.order_buffer[i])
            self.order_buffer[i].volume = 0
```

```
        self.order_buffer = [x for x in self.order_buffer if x.volume > 0]

    # DEPRECATED - DO NOT USE
    def place_order(self, order, tick):
        """
        puts an order into the trader buffer and then places the order with the exchange
        """
        self.order_buffer.append(order)
        self.exchange.place_order(order, tick)

class MarketMaker(Trader):
    def __init__(self, id, exchange, max_vol):
        Trader.__init__(self, id, exchange)
        self.max_vol = max_vol
        self.type = "market_maker"

    def run(self, order_prob, buy_prob, tick):
        # implement a check to see if the trader has enough account value to place a trade

        # update bid/ask via exchange

        # @todo: WRITE THESE FUNCTIONS
        self.best_bid = self.exchange.get_best_bid(gs.ASSET_CODE)
        self.best_ask = self.exchange.get_best_ask(gs.ASSET_CODE)

        self.update_buffer()

        self.cancel_orders(gs.MAKER_CANCEL_PROB)


        if random.uniform(0,1) < order_prob:
            # determine the size of the order
            vol = self.limit_volume_generator(self.max_vol)

            if random.uniform(0,1) < buy_prob:
                # determine the appropriate price to place the order at
                # uniformly distibuted some distance away from the best ask?
                # determine the price by folding the normal distribution

                norm_adjustment = abs(np.random.normal(loc = gs.MEAN, scale =
gs.STD_DEV))

                # log_price = random.uniform(-1*gs.L, np.log(self.best_ask))

                price = round(self.best_ask-norm_adjustment,3)

                # get the best ask
                # place a buy order
```

```
            order = Order(self.id+'_'+ str(self.order_num), vol, gs.ASSET_CODE, 1, 0, price,
self.id)
            self.call_exchange(tick, order)

            self.order_num +=1
         else:
            # determine the appropriate price to place the order at
            # uniformly distibuted some distance away from the best ask?
            if self.best_bid == 0:
               raise Exception("the best bid is 0")

            # fold the normal distribution
            norm_adjustment = abs(np.random.normal(loc = gs.MEAN, scale =
gs.STD_DEV))
            # log_price = random.uniform(np.log(self.best_bid), gs.L)

            price = round(self.best_bid+norm_adjustment,3)
            # place a sell order
            order = Order(self.id+'_'+str(self.order_num), vol, gs.ASSET_CODE, 1, 1, price,
self.id)
            self.call_exchange(tick, order)
      else:
         self.call_exchange(tick)

class RetailTrader(Trader):
   def __init__(self, id, exchange, max_vol):
      Trader.__init__(self, id, exchange)
      self.max_vol = max_vol
      self.type = "retail"

   def run(self, order_prob, buy_prob, tick):
      # implement a check to see if the trader has enough account value to place a trade

      # update bid/ask via exchange

      # @todo: WRITE THESE FUNCTIONS
      self.best_bid = self.exchange.get_best_bid(gs.ASSET_CODE)
      self.best_ask = self.exchange.get_best_ask(gs.ASSET_CODE)

      self.update_buffer()

      # this shouldn't do anything because this trader should only submit market orders
      self.cancel_orders(gs.RETAIL_CANCEL_PROB)

      if random.uniform(0,1) < order_prob:
         # determine the size of the order
         vol = self.limit_volume_generator(self.max_vol)

         if random.uniform(0,1) < buy_prob:
```

```
            price = gs.MARKET_BUY_LIMIT

            # get the best ask
            # place a buy order
            order = Order(self.id+'_'+ str(self.order_num), vol, gs.ASSET_CODE, 1, 0, price,
self.id)
            self.call_exchange(tick, order)

            self.order_num +=1
         else:
            # determine the appropriate price to place the order at
            # uniformly distibuted some distance away from the best ask?
            if self.best_bid == 0:
               raise Exception("the best bid is 0")

            price = gs.MARKET_SELL_LIMIT

            # place a sell order
            order = Order(self.id+'_'+str(self.order_num), vol, gs.ASSET_CODE, 1, 1, price,
self.id)
            self.call_exchange(tick, order)
      else:
         self.call_exchange(tick)
```

9.3 orderbook.py

```
"""Contains Orderbook class."""

from LIBRAmatcher import LIBRA_Buffer, LIBRAmatcher
from FCFSmatcher import FCFSmatcher
from order import Order
from matchedOrders import MatchedOrders
from globals import *

class Orderbook():
   """Will hold and store orders - might also contain the matching logic."""

   def __init__(self, name, mechanism_name = MATCHING_PROTOCOL):
      """Initialize the Orderbook."""
      self.name = name
      self.buffer_length = 1

      self.orderList = {0: [Order('placeholder1',1e10,self.name,1,0,0,'exchange')], 1e10:
[Order('placeholder2',1e10,self.name,1,0,1e10, 'exchange')]}

      self.best_bid=0
      self.best_offer = 1e10

      self.entry_buffer = []
```

```
    self.askPrices = set([1e10])
    self.bidPrices = set([0])

    if mechanism_name == 'FCFS':
        self.matcher = FCFSmatcher(self.buffer_length)
    elif mechanism_name == 'LIBRA':
        self.matcher = LIBRAmatcher()

# method that traders will use to place an order
def call_book(self, order_history, order = None):
    self.matcher.update_bests(self.best_bid, self.best_offer)

    if order:
        self.matcher.enter_order(order)

    orders_to_push = self.matcher.pull_down_orders()

    for order in orders_to_push:
        self.empty_order_from_buffer(order, order_history)
    orders_to_push = []

# used to push an order straight into the book e.g. during the initiation of the book
def place_order(self, order, order_history):
    """Adds an order to the entry buffer, that will then be processed by the exchange"""
    print('Manually firing an order into the orderbook')
    self.empty_order_from_buffer(order, order_history)


# DEPRECATED METHOD
# def old_place_order(self, order, order_history):
#     """Adds an order to the entry buffer, that will then be processed by the exchange"""
#     self.entry_buffer.append(order)
#     print("Adding an order from %s to the buffer"% order.trader_code)

#     len(self.entry_buffer)
#     if(len(self.entry_buffer)==self.buffer_length):
#         if MATCHING_PROTOCOL == 'FCFS':
#             print('Emptying buffer as it contains %d orders'%len(self.entry_buffer))
#             for order in self.entry_buffer:
#                 self.empty_order_from_buffer(order, order_history)
#             self.entry_buffer = []
#         else:
#             raise Exception('matching protocol has not been recognised')

def empty_order_from_buffer(self, order, order_history):
    #this is where the matching logic comes in
    #for each order in the buffer perform the matching protocol
    # for order in self.entry_buffer:
```

```
        # if the order is a buy order then check against existing ask prices
        if(order.dir == 0):
            # cycle until the order has been extinguished, either by
            # matching or by placing it into the book

            while order.price >= self.best_offer:

                best_offer_list = self.orderList[self.best_offer]

                while len(best_offer_list) > 0:
                    sell_order = best_offer_list[0]
                    if sell_order.volume <= order.volume:
                        # TODO: change the account values in the exchange? Potentially do it
                        # at the end of each simulation tick - this resembles clearing house
                        # behaviour in the real world

                        # append the match that just happened to the order history

order_history.append(MatchedOrders(order.trader_code,sell_order.trader_code,sell_order.pri
ce,sell_order.volume,order.code,sell_order.code))

                        order.volume-=sell_order.volume
                        best_offer_list.pop(0)
                        print("Matched %d units between buy order %s and sell order %s, execution
price %f" % (sell_order.volume,order.code,sell_order.code,sell_order.price))

                    else:
                        if sell_order.volume > order.volume:
                            sell_order.volume -= order.volume
                            order.volume = 0
                        else:
                            best_offer_list.pop(0)
                            order.volume = 0


order_history.append(MatchedOrders(order.trader_code,sell_order.trader_code,sell_order.pri
ce,order.volume,order.code,sell_order.code))
                        print("Matched %d units between buy order %s and sell order %s, execution
price %f" % (order.volume,order.code,sell_order.code,sell_order.price))
                        return 0

                # only do this when the above is not true
                # there are no longer any orders at the given price level
                self.askPrices.remove(self.best_offer)
                # find the new best offer price
                self.best_offer = min(self.askPrices)

        #if the code gets to here then place an order in the orderbook with the characteristics
of order
        # to get around the possibility that there is no defined list for the given price
```

```
        if order.volume > 0:
            try:
                self.orderList[order.price]
            except KeyError:
                self.orderList[order.price] = [order]
            else:
                self.orderList[order.price].append(order)


            self.bidPrices.add(order.price)
            # print(self.bidPrices)
            self.best_bid = max(self.bidPrices)

            print("Placed new buy order in the book - %d units at %f from
%s"%(order.volume, order.price,order.trader_code))
            return 0

        # if the order is a sell order then check against existing bid prices
        elif order.dir == 1:

            while order.price <= self.best_bid:
                # print(self.best_bid)
                best_offer_list = self.orderList[self.best_bid]

                while len(best_offer_list) > 0:
                    buy_order = best_offer_list[0]
                    if buy_order.volume <= order.volume:
                        # TODO: change the account values in the exchange? Potentially do it
                        # at the end of each simulation tick - this resembles clearing house
                        # behaviour in the real world

                        # append the match that just happened to the order history

order_history.append(MatchedOrders(buy_order.trader_code,order.trader_code,buy_order.pri
ce,buy_order.volume,buy_order.code,order.code))

                        order.volume -= buy_order.volume
                        best_offer_list.pop(0)
                        print("Matched %d units between buy order %s and sell order %s, execution
price %f" % (buy_order.volume,buy_order.code,order.code,buy_order.price))

                    else:
                        if buy_order.volume > order.volume:
                            buy_order.volume -= order.volume
                            order.volume = 0
                        else:
                            best_offer_list.pop(0)
                            order.volume = 0
                            print('testing')
```

```
order_history.append(MatchedOrders(buy_order.trader_code,order.trader_code,buy_order.pri
ce,order.volume,buy_order.code,order.code))
                    print("Matched %d units between buy order %s and sell order %s, execution
price %f" % (order.volume,buy_order.code,order.code,buy_order.price))
                    return 0

                # only do this when the above is not true
                # there are no longer any orders at the given price level
                self.bidPrices.remove(self.best_bid)
                # find the new best offer price
                self.best_bid = max(self.bidPrices)

            #if the code gets to here then place an order in the orderbook with the characteristics
of order
            if(order.volume > 0):
                try:
                    self.orderList[order.price]
                except KeyError:
                    self.orderList[order.price] = [order]
                else:
                    self.orderList[order.price].append(order)

                self.askPrices.add(order.price)
                self.best_offer = min(self.askPrices)


                print("Placed new sell order in the book - %d units at %f from %s"%(order.volume,
order.price,order.trader_code))
                return 0

    def printBuffer(self):
        for order in self.entry_buffer:
            order.print()

    # this is how an order can be removed from the book
    def cancel_order(self, order):
        if order.price in self.orderList.keys():
            if len(self.orderList[order.price])>0:
                self.orderList[order.price] = [x for x in self.orderList[order.price] if not x.code ==
order.code]
        self.matcher.cancel_order(order)

    def check_order_volume(self, order):
        matching_volume = 0
        if order.price in self.orderList.keys():
            same_price_orders = self.orderList[order.price]

            if len(same_price_orders) > 0:
                for book_order in same_price_orders:
```

```
                if book_order.code == order.code:
                    matching_volume+=book_order.volume
                    break

        matching_volume += self.matcher.check_order_volume(order)
        return matching_volume

    askPrices = set()
    bidPrices = set()
    orderList = {}
```

9.4 LIBRAMatcher.py

```
import globals as gs
import random

class LIBRA_Buffer(object):
    def __init__(self, type, direction, order, price = None, timer = gs.TIMER_LENGTH):
        # 0 is marketable, 1 is not marketable - ie. doesn't cross the spread
        self.type = type
        self.price = price
        self.orders = [order]
        self.timer = timer
        self.direction = direction

    def reduce_timer(self):
        self.timer-=1

    def add_order(self, order):
        self.orders.append(order)


class LIBRAmatcher(object):
    def __init__(self):
        self.buffers = []
        self.orders_to_push = []

    def enter_order(self, order):
        # algorithm 1 is here

        # if the order is a 'marketable buy'
        if order.dir == 0 and order.price >= self.best_offer:
            # find the marketable buy order buffer
            buffer = self.find_buffer(0, order.dir)
            # if there is a marketable buy order buffer, then append the new
            # order to the buffer
            if buffer:
                buffer.add_order(order)
```

```
            # otherwise create a new marketable buy buffer
            else:
                self.buffers.append(LIBRA_Buffer(0,order.dir,order))

        # if the order is a marketable sell
        elif order.dir == 1 and order.price <= self.best_bid:
            # find the marketable sell order buffer
            buffer = self.find_buffer(0, order.dir)
            # if there is a marketable sell order buffer, then append the new
            # order to the buffer
            if buffer:
                buffer.add_order(order)
            # otherwise create a new marketable sell order buffer
            else:
                self.buffers.append(LIBRA_Buffer(0,order.dir,order))

        # if the order is not marketable
        else:
            # find a matching buffer with direction, price and non-marketable
            buffer = self.find_buffer(1, order.dir, order.price)

            # if there is a matching non-marketable order then append the new
            # order to the buffer
            if buffer:
                buffer.add_order(order)

            # otherwise create a new non-marketable buffer with appropriate
            # characteristics
            else:
                self.buffers.append(LIBRA_Buffer(1,order.dir,order,order.price))

    # the method that gets called every tick
    def pull_down_orders(self):
        # create a new list of buffers which only contains orders with time
        # remaining
        buffers_with_time_remaining = []

        # put the checking code for the buffer timers here

        # this gets run every tick to check the timers of any current buffers,
        # if there are any with a zero then

        # shuffle the order of the buffers, so that no buffer has access
        # advantage
        random.shuffle(self.buffers)

        for buffer in self.buffers:
            # reduce the timer of the buffer under consideration
            buffer.reduce_timer()
```

```
if buffer.timer == 0:
    # algorithm 2 is here
    # create a dictionary of orders, keyed by participants
    participants_orders = dict()

    # iterate over the orders in the buffer
    for order in buffer.orders:
        # if the participant is already in the dictionary, append
        # the new order to their list of orders
        if order.trader_code in participants_orders.keys():
            participants_orders[order.trader_code].append(order)

        # otherwise add a new participant to the dictionary
        else:
            participants_orders[order.trader_code] = [order]

    # randomise the participant order with which orders will be
    # pulled off the pile
    participants = list(participants_orders.keys())
    random.shuffle(participants)

    # empty the participant orders dictionary
    while len(participants)>0:
        participants_copy = participants.copy()

        # cycle through the participants according to the randomised
        # list of participants
        for participant in participants:

            # if the participant still has orders to be pushed
            if participant in participants_orders.keys():

                # pop the first order
                order_to_go = participants_orders[participant].pop(0)

                # add it to the pushed list
                self.orders_to_push.append(order_to_go)

                # delete the participant from the dictionary if they
                # have no orders remaining
                if(len(participants_orders[participant]) == 0):
                    participants_copy.remove(participant)

        participants = participants_copy.copy()

    participants_orders = dict()

# if the timer still is not zero, then add it to the list which is
# used to redefine the buffers with time remaining
else:
```

```
            buffers_with_time_remaining.append(buffer)

        # change the list of buffers to only be buffers with time remaining
        self.buffers = buffers_with_time_remaining

        # push the list of orders which need to be pushed to the matching engine
        return self.orders_to_push


    # simple helper function to return a buffer that matches the conditions
    def find_buffer(self, buffer_type, direction, price = None):
        for buffer in self.buffers:
            if buffer.type == buffer_type:
                if buffer.direction == direction:
                    if buffer.price:
                        if buffer.price == price:
                            return buffer
        return None

    # needs to be called before each order is placed
    def update_bests(self, best_bid, best_offer):
        # update the best bid and offer in the matcher
        self.best_bid = best_bid
        self.best_offer = best_offer

        # at the start of each tick, clear the list of orders to be pushed to book
        self.orders_to_push = []

    # cancel function for an order in a LIBRA buffer - check buffers and then delete if the
order is in a buffer
    def cancel_order(self, order):

        # find the appropriate buffer that would contain the order
        buffer = None
        if order.dir == 0 and order.price >= self.best_offer:
            buffer = self.find_buffer(0,0)
        elif order.dir == 1 and order.price <= self.best_offer:
            buffer = self.find_buffer(0,1)
        else:
            buffer = self.find_buffer(1,order.dir,order.price)

        # if an appropriate buffer can be found then change the order list in the buffer
        # to one without the given order
        if buffer:
            buffer.orders = [x for x in buffer.orders if not x.code == order.code]

    def check_order_volume(self, order):
        matching_volume = 0
        for buffer in self.buffers:
            if buffer.price != order.price:
```

```
            pass
        else:
            for buff_order in buffer.orders:
                if buff_order.code == order.code:
                    matching_volume+=buff_order.volume
                    break
    for buff_order in self.orders_to_push:
        if buff_order.code == order.code:
            matching_volume+=buff_order.volume
            break

    return matching_volume
```