

Agent based modelling of a single-stock market on the JSE

Preyen Nair

A Dissertation submitted to the Faculty of Science, University of
the Witwatersrand, Johannesburg, in fulfilment of the
requirements for the degree of Master of Science.

Johannesburg, 2014

Declaration

I declare that this dissertation is my own unaided work. It is being submitted for the degree of Master of Science in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other university.

Preyen Nair

_____ day of _____ 2014 in _____

Abstract

The application of agent based modelling in finance allows market experiments to be undertaken which would normally be prohibitive due to cost, complexity and other factors. Agent based models use simple behaviour and interaction to produce complex outcomes. We introduce the requirements of an agent based market simulator based on protocol stipulated by the Johannesburg Stock Exchange. The requirements are then translated into a technical design. This design is implemented using the Microsoft .NET framework. The product of this design and creation approach is a market simulator which is then used to run three simulations where different agent behaviour is demonstrated. The approach and results of the simulations are documented to show possible use cases of the simulator.

Dedication

In memory of my grandmother
Salatchi K. Nair
1931 - 2013

Acknowledgements

I would like to thank my supervisor, Professor D. L. Wilcox. Her guidance, advice and teaching during this period has been invaluable in the conduct of this research.

Special thanks goes to my wife, Saiyuri, for her patience and support during this undertaking.

I would also like to acknowledge my parents, Mr and Mrs Y. M. Nair, who taught me that one can achieve anything through hard work and dedication.

Contents

1	Introduction	1
1.1	Financial exchanges	1
1.1.1	The limit order book	1
1.1.2	Price impact and dark pools	4
1.2	Agent-based modelling	6
1.2.1	Introduction	6
1.2.2	Why use ABMs	6
1.3	Simulating a single-stock market using computational agent-based modelling	7
1.4	Contributions of this work	8
1.4.1	Basic design	9
1.5	Research methodology	9
2	Requirements	10
2.1	Scope	10
2.2	Use cases	10
2.2.1	Place order	10
2.2.2	Cancel order	10
2.2.3	Amend order	11
2.2.4	Get order book data	11
2.3	Orders	12
2.3.1	Order types	12
2.3.2	Validity	13
2.3.3	Attributes	14
2.4	Order processing engine	14
2.4.1	Limit order	14
2.4.2	Market order	14
2.5	Order book data	15
3	Technical Design	17
3.1	Context overview	17
3.2	Data items	17
3.3	Matching engine	19
3.3.1	Limit Order Book and Order Logic	19

3.4	Communications module	19
3.4.1	Data feed	19
3.4.2	Trade interface	20
3.4.3	Asynchronous and synchronous operation	20
4	Implementation	22
4.1	Technology Overview	22
4.1.1	Microsoft .NET framework	22
4.1.2	Websockets	23
4.2	Dependency inversion	24
4.3	Data structures and algorithms	24
4.3.1	Data structures	25
4.3.2	Algorithms	26
4.4	Testing	29
4.4.1	Test cases	30
5	Showcase	38
5.1	The impact of heterogeneous trading rules	38
5.1.1	The model	39
5.1.2	Implementation considerations	42
5.1.3	Results	42
5.1.4	Interpretation of results	42
5.2	Zero intelligence agents	48
5.2.1	The model	48
5.2.2	Results	48
5.2.3	Interpretation of results	52
5.3	Agent learning and intra-day trading patterns	52
5.3.1	The model	52
5.3.2	Results and interpretation	56
6	Discussion	61
6.1	Conclusion	61
6.2	Limitations	61
6.3	Further work	61
	Appendices	63
A	Geometric Brownian Motion in C#	64
A.1	Background	64
A.2	Code	65
B	Code Samples	66
B.1	Example Test case	66
B.2	Random Liquidity Taker	69
B.3	Random Liquidity Maker	70
B.4	Heterogeneous trading rules agent	72

B.5	Intra-day trading patterns agent	77
B.5.1	Agent	77
B.5.2	Informed Agent	83
B.5.3	Uninformed Agent	84
B.5.4	Simulation coordinator	85
B.6	Standard limit order book implementation	92
B.7	SignalR Communications Module implementation	101
B.8	The Order data structure and enumerations	104

List of Figures

1.1	Quote-driven trading.	1
1.2	Bid-ask market.	2
1.3	Graphical representation of a limit order book.	3
1.4	Limit order book after a market order with quantity 60.	5
1.5	Obizhaeva and Wang (2013) observes that splitting large orders into N smaller orders minimizes execution cost. This diagram shows two large orders — one at the start of the period and the other at the end. Smaller orders are executed in between.	5
1.6	Number of dark pools and % trading volume (Schumpeter, 2011).	6
1.7	A typical agent based model. Note that agents may not necessarily be able to interact with all other agents (adapted from Macal and North (2010)).	7
1.8	The order book agent-based model.	9
2.1	Use case diagram.	11
2.2	Use case 1: Place order.	11
2.3	Use case 2: Cancel order.	12
2.4	Use case 3: Amend order.	12
2.5	Use case 4: Subscribe to data feed.	13
2.6	Limit order matching.	15
2.7	Market order matching.	16
3.1	Market simulator context overview.	17
3.2	Order data item.	18
3.3	Order update data item.	18
3.4	Match data item.	18
3.5	Limit order book and order logic definition.	19
3.6	Data Feed definition.	19
3.7	Trade Interface definition.	20
3.8	Asynchronous operation of the market simulation. Agents wait listening for changes in the order book and react immediately.	21
3.9	Synchronous operation of the market simulation. A controller process acts as an proxy between the limit order book and the agents. The controller process allows for the pooling of limit order book updates and actions of agents at discrete time steps.	21

4.1	The .NET Framework (Solis, 2012).	23
4.2	Communication between a SignalR server (left) and client (right).	24
4.3	An illustration of the Dependency Inversion Principle.	25
4.4	Data structures used to build the limit order book.	25
5.1	Time series data where all agents consider only their perfect information about the fundamental value of the simulated stock.	43
5.2	Time series data where only the noise component is taken into account. Agents do not consider the fundamental value.	43
5.3	Time series data where the fundamental component is taken into account but agents do not believe this information is perfect and this noise effects their decision making.	44
5.4	Time series data where both fundamental and chartist behaviour is taken into account as well as the effect of noise in decision making.	44
5.5	Limit order book snapshot at time step 2000 for final simulation.	45
5.6	Limit order book snapshot at time step 3000 for final simulation.	46
5.7	Limit order book snapshot at time step 4000 for final simulation.	46
5.8	Comparison variance of spot price where only the fundamental and noise components are considered and where all components are considered over a sliding window of size 10.	47
5.9	Best bid and ask time series with $p_\sigma = 0.5$.	49
5.10	Best bid and ask time series with $p_\sigma = 2$.	49
5.11	Best bid and ask time series with $p_\sigma = 10$.	49
5.12	Spread time series with $p_\sigma = 0.5$.	50
5.13	Spread time series with $p_\sigma = 2$.	50
5.14	Spread time series with $p_\sigma = 10$.	50
5.15	Return series with $p_\sigma = 0.5$.	51
5.16	Return time series with $p_\sigma = 2$.	51
5.17	Return time series with $p_\sigma = 10$.	51
5.18	Total profits generated by informed and uninformed agents under social learning.	55
5.19	Volume traded per trading session under social learning.	57
5.20	Coordination index per generation under social learning.	58
5.21	Volume traded per trading session under individual learning.	58
5.22	Coordination index per generation under individual learning.	59
5.23	Volume traded per trading session under random selection.	59
5.24	Coordination index per generation under random selection.	60

List of Tables

1.1	An example of a limit order book.	3
4.1	Number of operations required in the worst case when using linked lists versus using a sorted list of queues.	26
5.1	Decision table used by the agent where a_t^q and b_t^q are the best quoted ask and bid prices respectively.	41
5.2	Results for zero intelligence agent model.	52
5.3	Model parameters and base values.	53
5.4	Agent decision based on r_i , E_{id} , the current best bid and the current best ask.	54

Chapter 1

Introduction

1.1 Financial exchanges

1.1.1 The limit order book

Traditionally, clients were only able to buy assets in a market-making environment where large financial institutions post prices they are willing to buy and sell at. If a participant wished to trade at these market prices, an order is sent through and executed immediately. A participant wishing to trade had to trade immediately at the current market price. This is referred to as *quote-driven trading* and is illustrated in Figure 1.1. For example, a client will phone a large financial institution and request their buy and sell price for an asset. If he wishes to trade at those prices, he must conclude the transaction during the call or hang up otherwise.

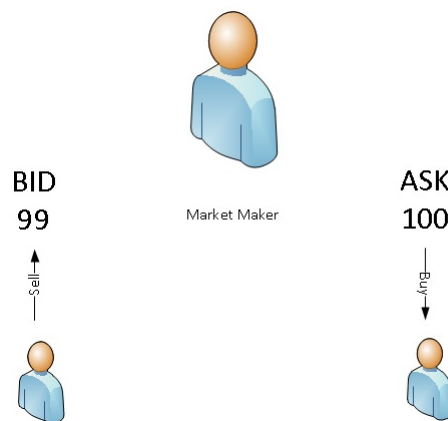


Figure 1.1: Quote-driven trading.

In more recent times, markets form a continuous double auction (Friedman, 1993). This is often referred to as a *bid-ask market* (Figure 1.2) where all participants are free to indicate prices they are willing to buy and sell at or trade immediately at the currency market price. More patient participants are able to make the market aware of the prices and quantity they are willing to trade at. Here, the participant is willing to accept the trade-off between execution time and a better price. These participants provide liquidity for future orders (Foucault et al., 2005) from traders wishing to execute their orders immediately. Older exchanges are run using an open outcry system where traders gather on an area of the trading floor (called the pit) and indicate their intentions using hand signals. The Chicago Mercantile Exchange (CME) still uses the open outcry System. See Chicago Mercantile Exchange for a manual on trading using hand signals. Modern exchanges are run electronically. The International Securities Exchange (ISE) and the InterContinental Exchange (ICE) are completely electronic exchanges and most major exchanges (NYSE, JSE, LSE) have introduced electronic trading mechanisms (Pirrong, 2003).

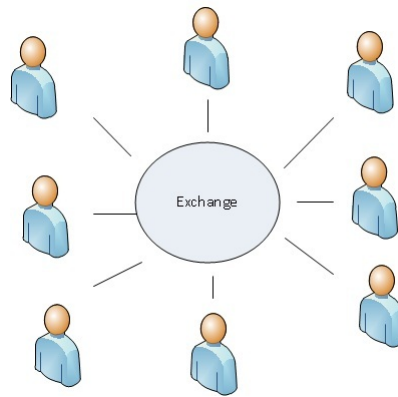


Figure 1.2: Bid-ask market.

Central to the bid-ask market is the *limit order book*. According to Gould et al. (2013), the limit order book mechanism matches buyers and sellers on over half the world's financial exchanges. Table 1.1 represents a limit order book. Figure 1.3 is a graphical representation of the same limit order book.

The limit order book is a record of all unmatched limit orders (see Section 1.1.1). For example, Figure 1.3 shows us two unmatched buy limit orders for price R98. One order is for a quantity of 40 and the other is for 10. The best buying price (bid) is R98 and the best selling price (ask) is R101. The difference between these two is the *bid-ask spread*.

Bids		Asks	
Quantity	Price	Quantity	Price
30	94		
50	96		
70	97		
50	98		
		30	101
		30	103
		20	104
		50	105

Table 1.1: An example of a limit order book.

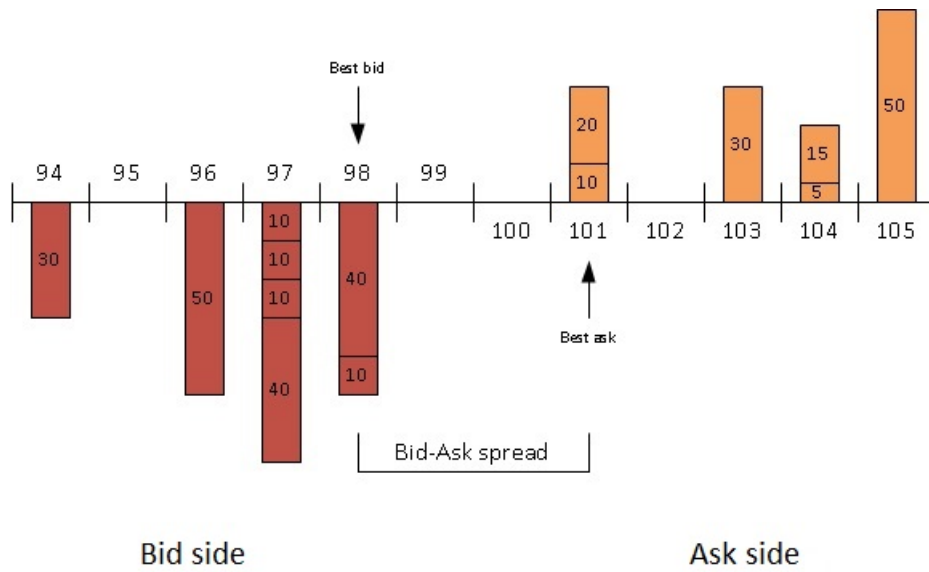


Figure 1.3: Graphical representation of a limit order book.

Order types

Limit orders are placed by market participants who are not simply willing to trade at the current market price but wish to indicate their price limits. A sell limit order (ask) indicates the lowest price the participant is willing to sell at. Conversely, a buy limit order indicates the highest price the participant is willing to buy at. Along with the associated price, the quantity a participant is willing to trade is also included. Limit orders are not executed immediately and may not be executed at all.

Market orders are placed by market participants who wish to trade instantly at the prevailing market prices. Buy market orders and sell market orders are used by participants willing to trade at the currently available price(s).

Order matching

Limit orders are stored in a First-In-First-Out (FIFO) manner and matched accordingly. When a market order arrives, it is matched against limit orders with the best price. If the quantity available at that price is not sufficient, the market order is further matched against the second best price and so on until the full quantity is satisfied. For example (referring to Figure 1.3):

A sell market order arrives with a quantity of 60. The two orders at R98 are matched. This leaves a quantity of 10 unmatched on the market order. Now, the first order for R97 gets matched. Hence, the market order with quantity 60 has been matched at 50 on R98 and 10 on R97. The order book is now in the state shown in Figure 1.4. The best bid is now R97 and the bid-ask spread has widened to R4 (R101 - R97).

1.1.2 Price impact and dark pools

Easley and O'hara (1987) observes that falls in asset prices after large sell orders and rises in asset prices after large buy orders can be attributed to the information affect. The market could perceive large orders to be the effect of an informed trader with private information. Further work has been conducted on quantifying price impact (Potters and Bouchaud, 2003) as well as finding optimal liquidity strategies for institutional trades. Obizhaeva and Wang (2013) shows that splitting large trades as shown in Figure 1.5 minimizes their total execution cost.

Exchanges have introduced a mechanism dubbed 'dark pools' to order books where no information about any participants' trading intentions is made publicly available (Gould et al., 2013). Kratz and Schöneborn (2013) says that while transparent (traditional) exchanges are open to price impact, dark pools admit execution uncertainty due to rules around minimum order sizes and execution. They go on to investigate an optimal liquidity strategy which involves trading on both dark pools and classic (transparent) markets.

According to Schumpeter (2011), in 2010, dark pools accounted for 12.5% of trading volume in the US and 10% in Europe. See Figure 1.6.

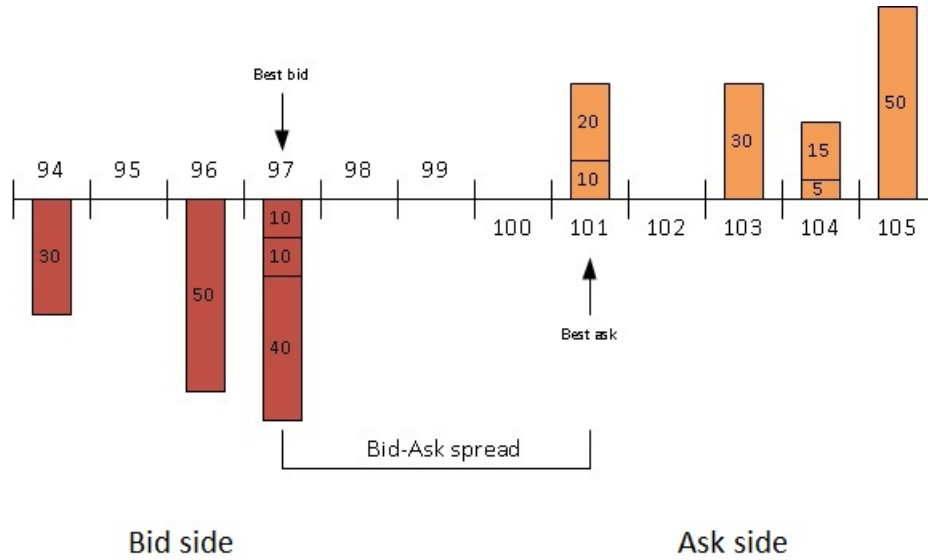


Figure 1.4: Limit order book after a market order with quantity 60.

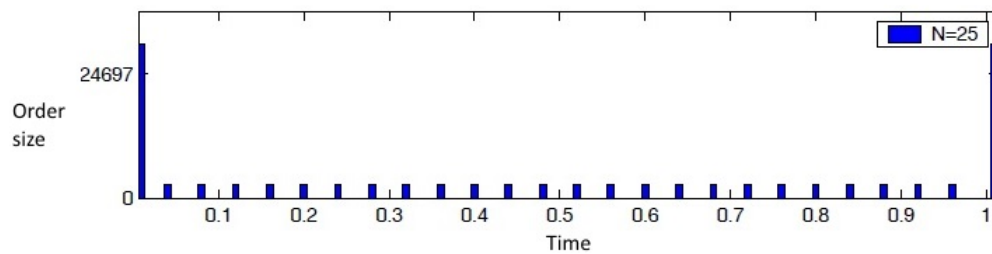


Figure 1.5: Obizhaeva and Wang (2013) observes that splitting large orders into N smaller orders minimizes execution cost. This diagram shows two large orders — one at the start of the period and the other at the end. Smaller orders are executed in between.

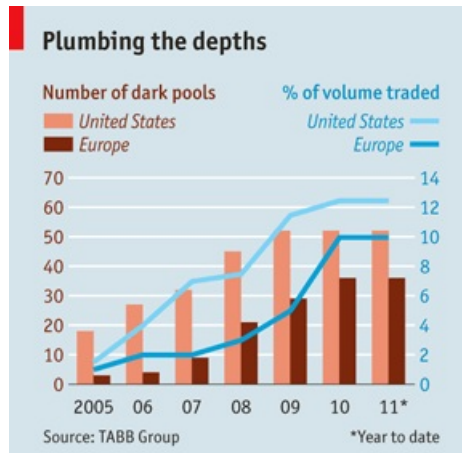


Figure 1.6: Number of dark pools and % trading volume (Schumpeter, 2011).

1.2 Agent-based modelling

1.2.1 Introduction

Agent-based modelling (ABM) is a relatively new method of modelling behaviour in complex systems. Schelling (1971) discusses one of the earlier applications in social science. Recent increases in computing power has led to a spurt of applications across a wide variety of fields. These include natural resource planning (Anderson and Evans, 1994; Soulié and Thébaud, 2006), ecology (Cao et al., 2009; McLane et al., 2011), city management (Gao et al., 2012), military planning (MacKenzie et al., 2012), economics (Roosmand et al., 2011), sustainable development (Zhang et al., 2011a,b) and finance (Yang, 2002).

According to Macal and North (2010), an agent-based model has three elements:

- A set of (possibly heterogeneous) self-contained, autonomous, agents with attributes and behaviour.
- Relationships between agents and methods of interaction (how and with whom agents can interact).
- An environment agents can interact with.

Figure 1.7 illustrates a typical agent based model.

1.2.2 Why use ABMs

Castiglione (2006) states that computational methods are useful in the following circumstances:

- The problem is too difficult to solve analytically.

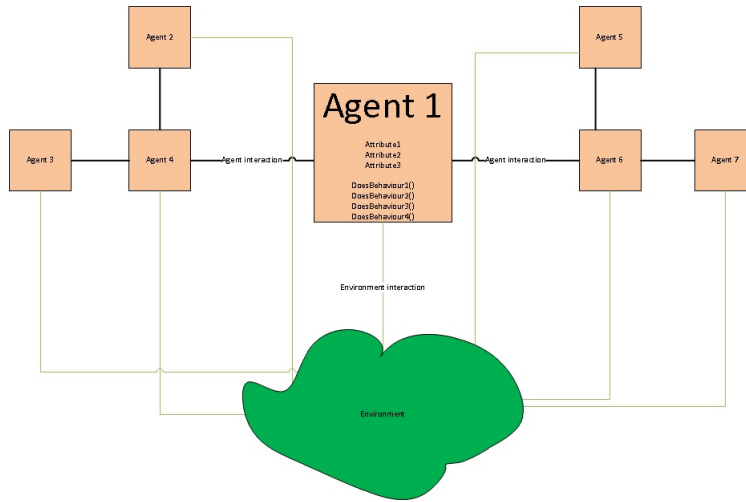


Figure 1.7: A typical agent based model. Note that agents may not necessarily be able to interact with all other agents (adapted from Macal and North (2010)).

- A theoretical result needs to be confirmed.
- It is not feasible to perform an experiment in the real environment.¹

Agent-based models provide a platform to derive evidence empirically in a simulated environment. This evidence can be used to confirm (or disprove) theoretical results. Rich results are obtained from the interaction of simple agents. This simplicity allows for rapid development of agent-based models.

1.3 Simulating a single-stock market using computational agent-based modelling

As mentioned in Section 1.2.2, agent-based models help us create a verified, realistic environment to find solutions to problems where it is not feasible to perform experiments in the real environment or the solution cannot be expressed analytically. Given limited funds, testing theories and strategies in financial markets can often prove to be difficult and expensive. Agent-based models have been used to test behaviours in financial markets.

Marks (1992) is one of the early works using computational methods, game theory in this case, to perform simulation of markets. Tournaments of repeated games were run where it was observed that market participant cooperation was apparent in oligopolies. Marks went on to say that, while computational power was limited at the time, there was no conceptual limit to the number of players

¹It is important to note that the model must first be validated against a real environment. This is done by comparing the model's output with real data.

in the game and the complexity of problems which could be solved computationally would increase along with computing power.

The Sante Fe Artificial Stock Market (Arthur, 1996) is a simulator used to run experiments in a controlled manner. By design, many of its features can be changed to allow researchers to test a wide variety of theories. Many examples follow this model where agents are programmed with different behaviour rules and run in a market environment to produce stock market data. The properties and patterns in this data can then be compared against a real stock market. For example, Chan et al. (1999) went on to validate an agent-based model against a human-based experimental market. Their findings show that the agent-based model was able to produce similar observables (price, market efficiency, bid-ask spread, etc.) to the experimental market. Bak et al. (1997) describes a model consisting of ‘fundamental’ traders who make decisions based on analysis of the asset and ‘noise’ traders who analyse only the market dynamic. Tay and Linn (2001) extended the work by Arthur (1996) and argued that real traders cannot follow a vast array of strict rules while trading. They went on to show that realistic results can be achieved by using a genetic-fuzzy mechanism to guide decision making. Izumi and Ueda (2001) interviewed dealers to gather data on dealer interaction and learning in a foreign exchange market. They developed an artificial market mimicking this agent interaction as well as a genetic learning mechanism and showed that realistic market behaviour (peak and fat-tailed distribution of price changes) could be explained by these mechanisms. Nicolaisen et al. (2001) reports on a computational experiment run on a simulator where electricity prices are determined by a double auction. In their model, agents make decisions based on individual Roth Erev learning. This is later compared to agents learning using social genetic algorithms. They show that their model produces realistic market microstructure characteristics and that the market microstructure strongly determines agent behaviour — agent learning methods are less relevant. Maslov (2000) introduced a simple limit order-driven model and was able to reproduce a market which showed realistic features such as fat-tailed price fluctuation distributions using purely random buy/sell decisions (no strategies). Raberto et al. (2001), Cont (2007) and Qi (2009) showed that agent-based models can exhibit further key financial market characteristics such as volatility clustering. Chen et al. (2010) built an agent based model and used it to test optimal liquidation strategies.

1.4 Contributions of this work

Previous work has mainly focused on the markets in developed economies. This research will add to the literature by documenting the specification, design, implementation and verification of an agent-based limit order book model which runs according to protocol stipulated by the JSE equities market (Johannesburg Stock Exchange, 2008, 2010). Particular attention was paid to the idiosyncrasies specific to the JSE such as order types, expiry instructions, partial matching and cancellations. The bulk of this analysis can be seen in Section 2 where JSE

specific requirements are documented. The outcomes also include a detailed design of the simulator which has often been neglected. To prove its use as a workbench, the simulator itself was used to implement three agent based financial market models. The outputs of these models are analysed and tested for realism.

1.4.1 Basic design

Figure 1.8 shows a basic outline of the model. Agents represent the market participants and are heterogeneous in that each can have a different strategy behind its decision making. The environment represents the limit order book. Agents affect the limit order book by placing different types of orders or by cancelling orders. Interaction with the limit order book must be bi-directional to allow for agents to be notified when orders are matched or expire.

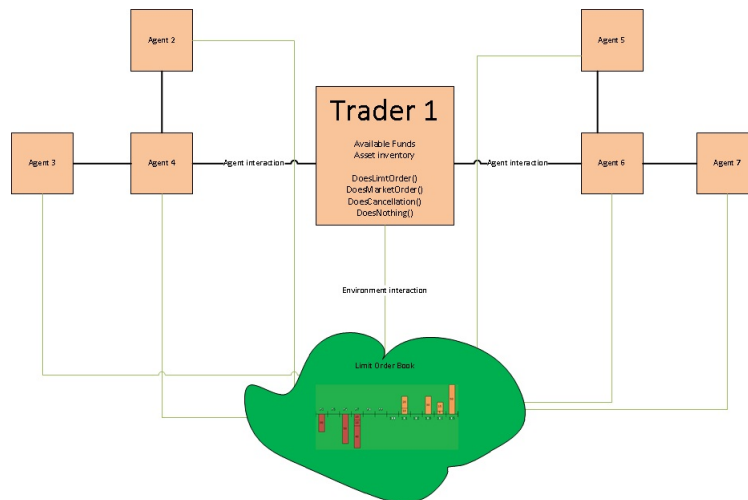


Figure 1.8: The order book agent-based model.

1.5 Research methodology

This research was carried out with a *design and creation* approach. A design phase was carried out where requirements were stated (Chapter 2) and a technical design presented (Chapter 3) for implementation (Chapter 4). The outcome of the implementation phase (the simulator) was tested to make sure the results are reliable. Experiments were then run with different agent models and documented to showcase the simulator (Chapter 5).

Chapter 2

Requirements

2.1 Scope

This simulator includes the standard order types and mechanisms associated with a continuous double auction market as specified in Johannesburg Stock Exchange (2010, 2008). Particular attention has been paid to ensuring the simulator is able to satisfy the requirements brought about by the specific JSE protocol outlined in these documents. Specifically excluded were auction periods and hidden orders.

2.2 Use cases

Use cases identify the types of interactions in a system and the actors involved (Sommerville, 2004a). Figure 2.1 illustrates the use cases for the market simulator. Each use case is explained using sequence diagrams. Sequence diagrams are dynamic models which show the order and detail each interaction in a use case (Sommerville, 2004b).

2.2.1 Place order

1. Trader sends order through with required attributes (see Section 2.3.3).
2. The order is processed at the exchange (see Section 2.4).
3. The trader receives the result containing the Trade ID, any matches or errors.
4. If the order has resulted in any matches, the other counterparty is notified.

2.2.2 Cancel order

1. The trader sends through a cancellation request with the relevant order ID.

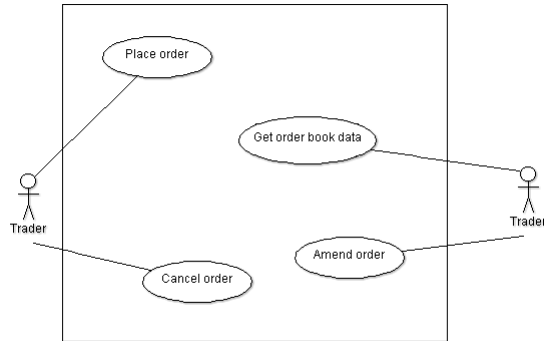


Figure 2.1: Use case diagram.

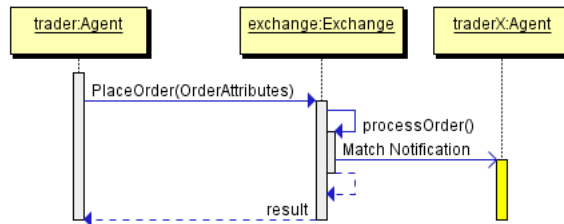


Figure 2.2: Use case 1: Place order.

2. The order is removed from the limit order book.
3. The trader receives the result and errors (if any).

2.2.3 Amend order

1. The trader sends through an order amendment with the relevant trade ID and order attributes.
2. Amendments result in the loss of time priority. This means the amended order is processed as a new order.
3. The order is processed at the exchange.
4. The trader receives the result containing the Trade ID, any matches or errors.
5. If the order has resulted in any matches, the other counterparty is notified.

2.2.4 Get order book data

1. The trader sends through a request to subscribe to the limit order book data feed.

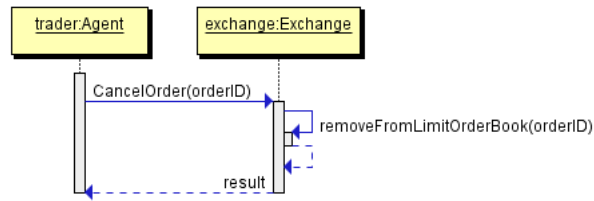


Figure 2.3: Use case 2: Cancel order.

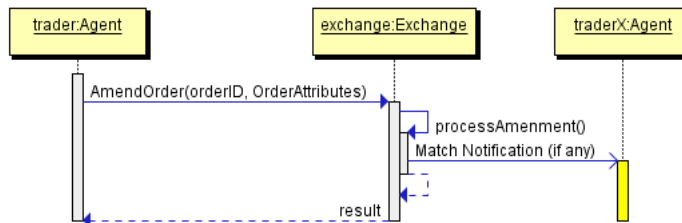


Figure 2.4: Use case 3: Amend order.

- Whenever a change in the order book occurs, the trader is notified by the exchange.

2.3 Orders

2.3.1 Order types

A market participant can send through many different types of trade orders. The types of orders specified by JSE protocol in Johannesburg Stock Exchange (2010, 2008) are explained in this section.

Limit order

Limit orders are placed by market participants who are not simply willing to trade at the current market price but wish to indicate their price limits. A sell limit order indicates the lowest price the participant is willing to sell at. Conversely, a buy limit order indicates the highest price the participant is willing to buy at. Along with the associated price, the quantity a participant is willing to trade is also included. Limit orders are not executed immediately and may not be executed at all. Limited orders are executed in price-time priority where the best prices are executed first in the order of arrival.

Market order

Market orders are placed by market participants who wish to trade instantly at the prevailing market prices. Buy market orders and sell market orders are

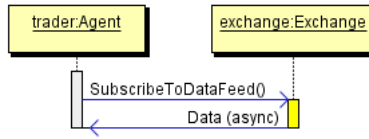


Figure 2.5: Use case 4: Subscribe to data feed.

used by participants willing to trade at the currently available price(s).

Stop order

Stop orders are market orders with a specified *stop price*. Once this price is reached, it becomes a regular new market order.

Stop limit order

Limit stop orders are limit orders which also specify a *stop price*. Once this price is reached, it becomes a regular new limit order.

2.3.2 Validity

When a participant enters an order, it indicates when an order is valid and for how long it is valid for. According to the JSE protocol, validity can be time based or period based. Orders with a time based validity are removed (unless executed or cancelled) after a specified data or time.

- Good Till Time (GTT) — valid until a specified expiry date.
- Good Till Cancelled (GTC) — expiry is set to the maximum for the instrument (currently 90 days).

Period based validity enables the ‘parking’ of orders when orders are queued and injected during a specific trading period.¹

- Good for day (GFD) — Only valid during continuous trading period.
- At the Open (ATO) — Only valid during the next opening auction.
- Good for Auction (GFA) — Valid for the next available auction.
- Good for Intra-Day Auction — Valid for next scheduled intra-day auction.

Execution based validity specifies what should happen to orders immediately submitted to the book after entry by the participant.

- Execute and Eliminate — Partial matching allowed. Unexecuted volume is removed from the book.

¹Since auction periods are out of scope, period validity does not apply to this project.

- Force or Kill — Either match all volume or nothing. The entire order is revoked if not completely filled.

2.3.3 Attributes

Field	Required	Description
Side	Yes	Buy or sell.
Order type	Yes	See Section 2.3.1.
Time in force	No	The duration the order is valid for. (Default: Day)
Quantity	Yes	Number of units of the asset being bought or sold.
Price	No	The execution price of the order. Required for Limit and Stop Limit orders.
Stop Price	No	The price at which the stop order is entered as a regular order. Only required for stop orders
Expiry Time	No	An order with 'Good Till Time' validity will expire at this time on the current day
Expiry Date	No	An order with 'Good Till Day' validity will expire on this date

2.4 Order processing engine

2.4.1 Limit order

Limit orders can be matched against other limit orders.

1. A buy (sell) limit order arrives.
2. The order is matched against any sell (buy) limit orders which have a lower (higher) price.
3. For each match, a notification is sent to the market participant who placed the order.
4. The unmatched remainder is added to the order book at the end of the queue at the indicated price.

Figure 2.6 depicts this process graphically.

2.4.2 Market order

1. A buy (sell) market order arrives.
2. The matching begins with the first order which arrived at the best available price.

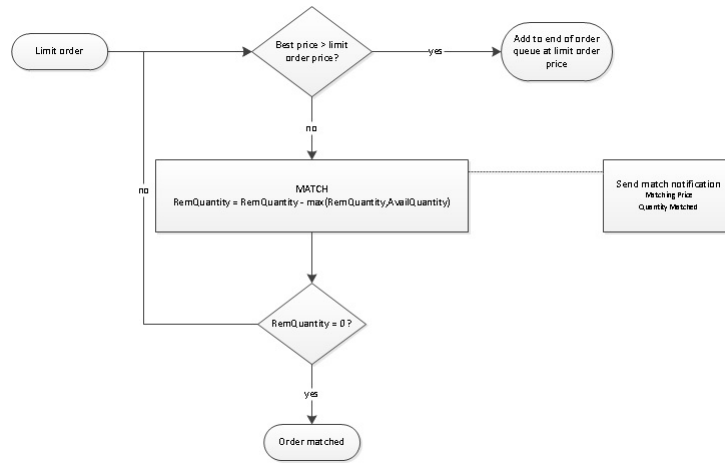


Figure 2.6: Limit order matching.

3. If the quantity of this order is not sufficient, the matching moves on to the next order. If there are no orders available at the current price, the matching moves to the next best price.
4. For each match, a notification is sent to the market participant who placed the order.
5. This process continues until the arriving order is fully matched or there are no more orders available.

See Section 1.1.1 for an example. Figure 2.7 depicts this process graphically.

2.5 Order book data

All market participants must be able to receive snapshots representing the current state of the limit order book. They can do this either on request or on a subscription basis where they receive the current snapshot after every change in the limit order book. An example is illustrated in Table 1.1.

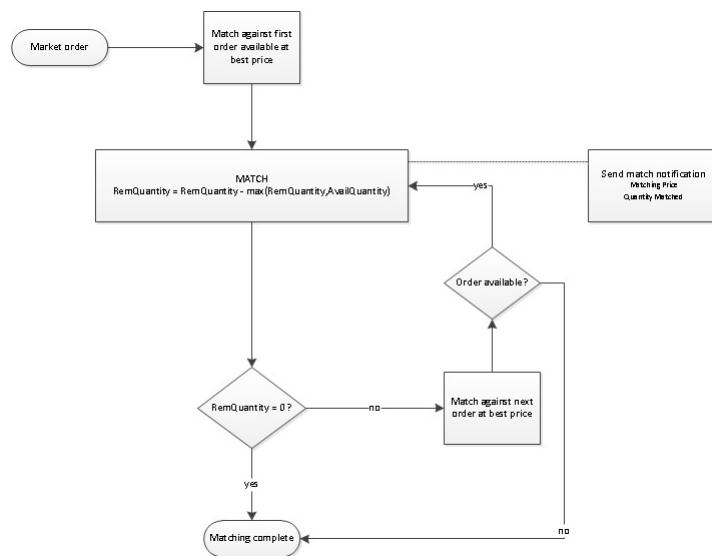


Figure 2.7: Market order matching.

Chapter 3

Technical Design

3.1 Context overview

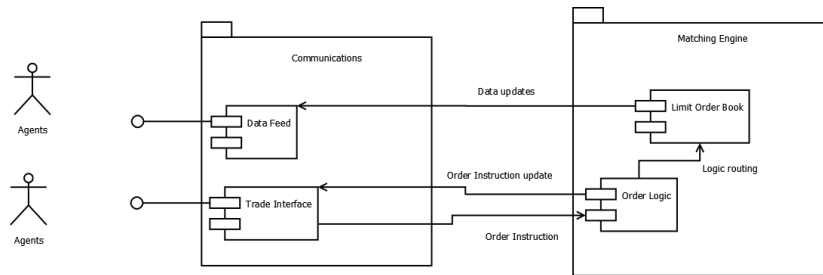


Figure 3.1: Market simulator context overview.

Figure 3.1 gives an overview of the components which will make up the market simulator as well as their interactions. Agents communicate with the market via the Communications Module. Order instructions are sent to the Trade Interface. The Trade Interface will also reply with updates to the order instruction. These updates will include execution results and errors (if any). The Trade Interface passes order instructions to the order logic component in the Matching Engine module. This component inspects the order and routes it to the appropriate function on the Limit Order Book (LOB) component. The LOB is where all order matching takes place. Any updates to the LOB are communicated via the Data Feed component. Order instruction results are returned to the Order Logic component.

3.2 Data items

The Order and Order Update data items are depicted by Figures 3.2 and 3.3 respectively. The Order data item contains the attributes set out in the re-

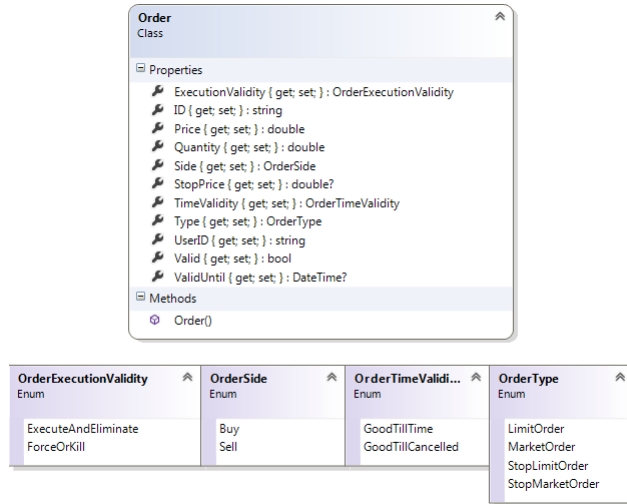


Figure 3.2: Order data item.

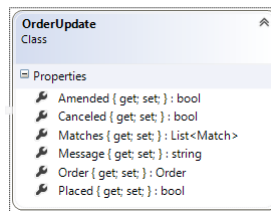


Figure 3.3: Order update data item.

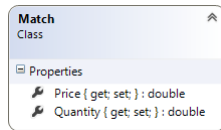


Figure 3.4: Match data item.

quirements in Section 2.3.3. Agents will send an instance of this item to the Trade Interface. The Order Update data item is the result of executing the order. It contains the original order instruction, flags indicating success and a list of matches (if applicable). The Order Update also has a field for any error messages.

3.3 Matching engine

3.3.1 Limit Order Book and Order Logic

Order routing logic is triggered by the OnOrder event on the Trade Interface (Figure 3.7). Here, the order instruction is inspected and is routed to the appropriate method. Market orders are routed to the *ProcessMarketOrder* method while limit orders can either be added (*ProcessLimitOrder*), amended (*AmendLimitOrder*) or cancelled (*CancelOrder*). The logic for processing orders is laid out in Section 2.4.

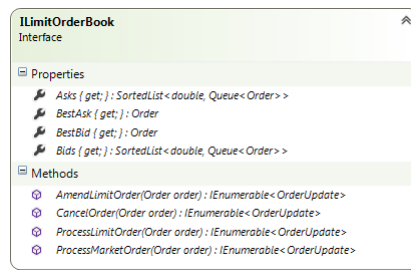


Figure 3.5: Limit order book and order logic definition.

3.4 Communications module

All communication between agents and the simulator is facilitated by the Communications module. Communication can be divided into two parts: the data feed and the trade interface.

3.4.1 Data feed

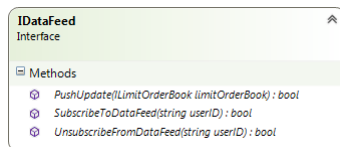


Figure 3.6: Data Feed definition.

Agents can *subscribe* to and *unsubscribe* from the data feed. This enables them to be notified of any changes in the limit order book. These notifications are triggered by the Limit Order Book component on every update by calling the *PushUpdate* method.

3.4.2 Trade interface

Agents create, amend and cancel trade requests via the Trade Interface. They do this by sending an order instruction to the *ProcessOrderInstruction* method. Execution results are pushed back to agents when the Order Logic component calls back to the *PushOrderInstructionUpdate* method while providing an order update.

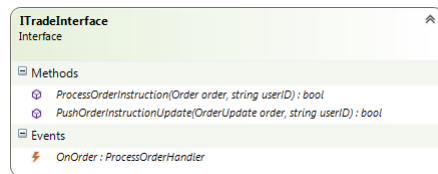


Figure 3.7: Trade Interface definition.

3.4.3 Asynchronous and synchronous operation

The simulator deploys Discrete Event Simulation (DES). Nelson et al. (2001) describes DES as simulations where models are analysed by numerical methods instead of analytical methods. The output from these numerical methods generate an artificial history of the system. This history is gathered and analysed to estimate real-world behaviour. The simulator represents the world as a limit order book and agents with their own attributes. A report of price movements and changes in agent attributes is generated while passing through market updates which represents the events. The delivery of events to agents can either be asynchronous or synchronous. The Communications module has been designed to operate asynchronously: agents wait listening for updates on execution results and changes in market data instead of making calls to get updates and the latest market data. This translates to fewer calls being made to the Communications module. This also allows simulations to operate in real-time — agents can make decisions immediately once an event is raised to indicate a change in the order book. This is depicted in Figure 3.8. Asynchronous operation is advantageous when trying to simulate features of a market which are only apparent using continuous agent models. However, there may be reasons for the simulation to be synchronous where a controlling process polls agents at discrete time steps for their decisions. An example of this would be a model where agents believe the fundamental value of an asset follows a stochastic process which is solved numerically. This mode of operation is supported by the simulator. Figure 3.9 shows a mechanism for achieving synchronous behaviour from agents when the Communications module operates asynchronously. A controller process acts as

a store for the limit order book updates and is a driver for the simulation. The controller process polls every agent for its next action using its latest limit order book update and executes on the agents' behalf.

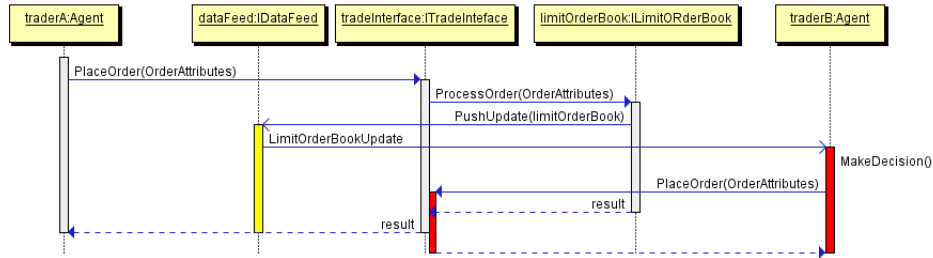


Figure 3.8: Asynchronous operation of the market simulation. Agents wait listening for changes in the order book and react immediately.

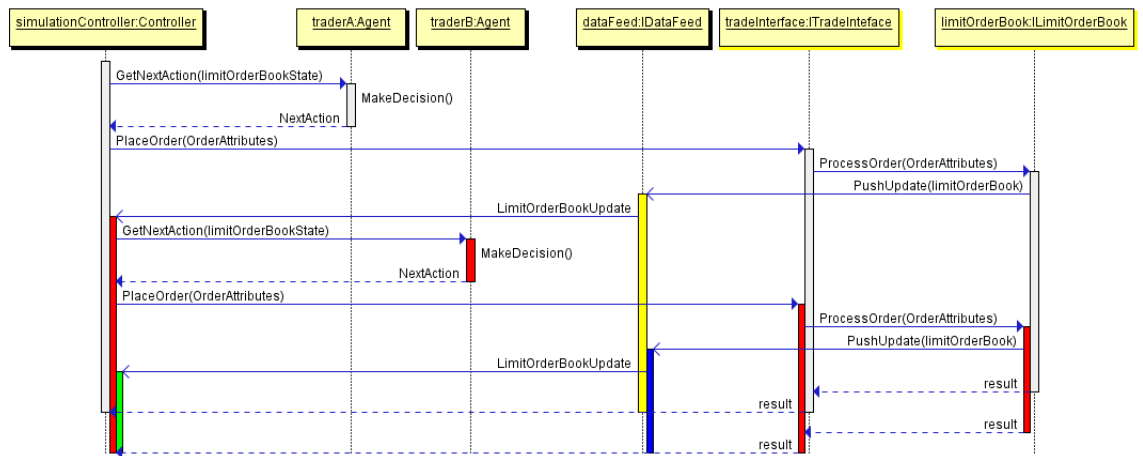


Figure 3.9: Synchronous operation of the market simulation. A controller process acts as a proxy between the limit order book and the agents. The controller process allows for the pooling of limit order book updates and actions of agents at discrete time steps.

Chapter 4

Implementation

4.1 Technology Overview

The market simulator was implemented using Microsoft's .NET framework (Microsoft, 2013). The chosen language is C#. While the .NET framework mainly targets the Microsoft Windows platform, alternative implementations such as Mono (Xamarin, 2013), a free and open source implementation of the .NET framework, allow the simulator components to run on multiple platforms. Web sockets facilitate the underlying communication mechanism between agents and the market simulator. The SignalR (SignalR, 2013) web sockets implementation is used in the simulator.

4.1.1 Microsoft .NET framework

The .NET framework provides a software development environment which abstracts operating system and hardware concerns from the user. It supports multiple languages such as C#, Visual Basic and Visual C++. As illustrated in Figure 4.1, the framework is split into the following compile time and run time components:

- Source code — programs created by the user.
- Compilers — A multitude of compilers transform user programs into Common Intermediate Language (CIL) assemblies.
- Common Intermediate Language — All .NET programs are compiled into CIL for interpretation.
- Common Language Runtime (CLR) — The CLR provides the abstraction layer between the operating system and the CIL. Here, CIL code is interpreted by a Just In Time (JIT) compiler into native code to be executed directly by the operating system.

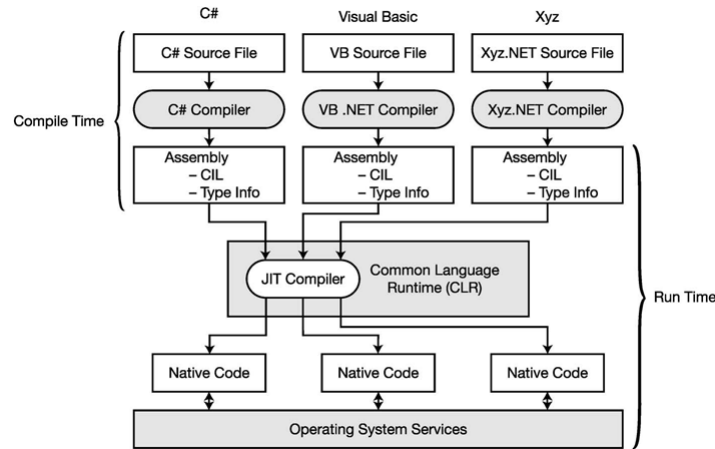


Figure 4.1: The .NET Framework (Solis, 2012).

Due to this design, the .NET Framework’s CIL allows multiple languages to be combined into a single program while the CLR allows this program to be executed on different platforms. Currently, the most supported platforms are Microsoft’s Windows, Windows Server and Windows Phone. However, there are implementations for other platforms.

4.1.2 Websockets

Traditional web services allow single directional communication where a client typically calls a server with a request and the server responds with a result. Websockets allows bi-directional communication to occur where not only can a client execute code on the server but the server can execute code on the client. This mechanism was initially simulated by using long polling where a client would send a request and wait on that request until the server requires client side execution. This was seen as abuse of the HTTP protocol and the websockets protocol was introduced where a simple TCP connection is used for bi-directional communication (Hickson, 2010). This makes it the natural technology choice for communication in the market simulator: clients send order instructions to the server but the server also sends order updates and limit order book updates to the clients.

SignalR

SignalR is a web sockets implementation for Microsoft ASP .NET — Microsoft’s web technology. A SignalR process requires a web server. This is provided by an OWIN container (OWIN Project) which allows a web server to run in any .NET process. Figure 4.2 shows the communication pattern between a SignalR

server and client. The client first initializes a connection to the server. Once a connection is initialized, either the client or server can invoke functionality on the other. The ability for the server to invoke functionality on the client is the key to allowing the asynchronous functionality required for the Communications Module described in Section 3.4.3. See Section B.7 for the Communications Module implementation using SignalR.

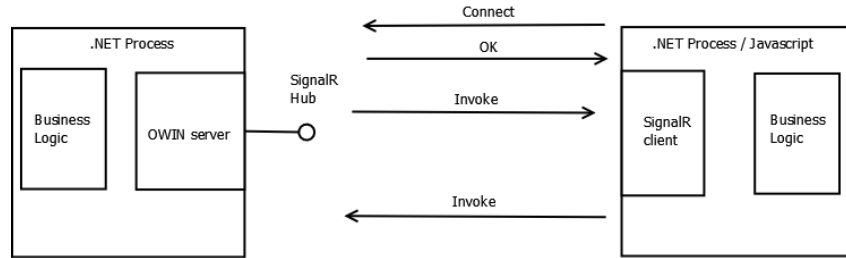


Figure 4.2: Communication between a SignalR server (left) and client (right).

4.2 Dependency inversion

Martin (1996) describes the Dependency Inversion Principle as

- High level modules should not depend upon low level modules. Both should depend upon abstractions, and
- abstractions should not depend upon details. Details should depend upon abstractions.

Using C# Interfaces, the programmer is able to set up contracts (abstractions) between the different components of the simulator. Each component relies on its dependencies to fulfil a certain contract. These sub components then provide the implementation of those contracts. This principle gives us the ability to easily change contract implementations without changing the components which rely on them. Figure 4.3 illustrates the principle using a UML diagram. Here, the Communications Module depends on two abstractions — the `IDataFeed` and `ITradeInterface` interfaces. Those two abstractions are detailed by the `DataFeed` and `TradeInterface` implementations respectively.

4.3 Data structures and algorithms

Performance is a major factor to consider when implementing a simulator. For a market simulator, the main bottleneck is the limit order book. Careful consideration must be made with regards to the data structures used to model the limit order book as well as the algorithms operating on them.

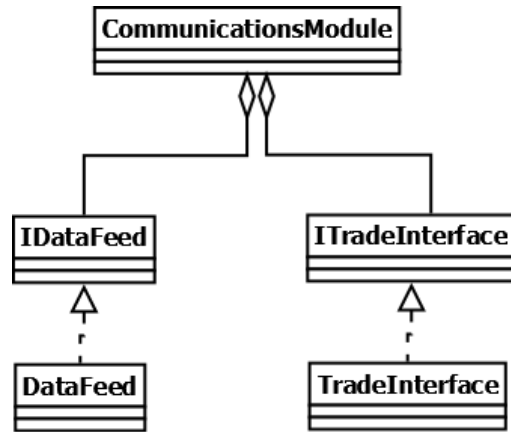


Figure 4.3: An illustration of the Dependency Inversion Principle.

4.3.1 Data structures

The limit order book has been split in to two sorted lists — one to store buy limit orders (bids) and the other to store sell limit orders (asks). Each element in the sorted lists represents a price. The price element itself is a queue containing orders at that price. Figure 4.4 illustrates this. It is important to note that the sorted list is not a linked list but one with an implementation similar to Microsoft (2014) which makes use of arrays internally. Arrays allow for elements to be accessed directly by index instead of traversing a list. This is a requirement for implementing a bisection search which requires $\log_2 n$ comparisons in the worst case.

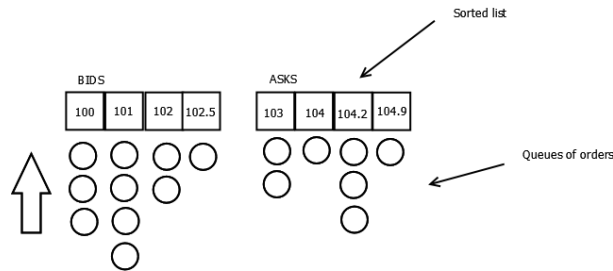


Figure 4.4: Data structures used to build the limit order book.

A queue structure was chosen for an order as its First-In-First-Out property caters for the time priority requirement. The next order to be matched is popped off the front of the queue while new orders are pushed onto the end of the queue with the appropriate price. When using a queue, getting the oldest order at a certain price (for matching) and inserting a new order at a certain price both require only one operation in all cases. Locating the queue which represents the

Operation	Linked Lists	Sorted list of queues
Locating the desired price	n	$\log_2 n$
Adding a new price	n	$\log_2 n$
Inserting a new order	n	1
Getting the time prioritized order	1	1

Table 4.1: Number of operations required in the worst case when using linked lists versus using a sorted list of queues.

desired price requires $\log_2 n$ comparisons at most as they are stored in a sorted list where n is the number of distinct prices. Inserting a queue at a new desired price also requires $\log_2 n$ comparisons at most where n is the number of distinct prices. Using these data structures allow for natural and efficient tracking of time and price priority.

Comparison with linked lists

Section 4.3.1 shows that using sorted lists and queues are efficient and effective when modelling a limit order book. Using normal linked lists may make the implementation less complex. However, this is inefficient. When using linked lists, inserting an order at a certain price requires n comparisons in the worst case where n is the number of orders currently at that price. Locating the list containing the desired price also requires n comparisons in the worst case where n is the number of distinct prices. This is the same for adding new orders at a price which doesn't currently exist in the order book. Table 4.1 summarizes the differences between using linked lists and using a sorted list of queues. We believe the small increase in implementation complexity is worth the gain in efficiency.

The Order data structure

The order data structure is implemented as a standard class in C# with properties. Properties which represent a range of options — validity and order side for example — are implemented as enumerations. The code for these data structures can be found in Section B.8.

4.3.2 Algorithms

In this section, we will discuss the main operations of a limit order book: adding limit orders and executing market orders.

Adding a limit order

The code for adding a new sell limit order is listed below. The code for adding a new buy limit order is similar. We first search the sorted list of prices for the new limit order price. Due to the list being sorted, this operation requires

$\log_2 n$ comparisons in the worst case where n is the number of distinct prices. If the price is not found, we insert a new queue at the limit order price. Inserting into an ordered list also requires $\log_2 n$ comparisons in the worst case. We then insert the new order. Due to the insertion operation acting on a queue, only one operation is required. We can see that the complexity of adding a new limit order is $O(\log n)$.

```

1  if (!Asks.Keys.Contains(order.Price))
2  {
3      Asks.Add(order.Price, new Queue<Order>());
4  }
5
6  Asks[order.Price].Enqueue(order);
7
8  return new[]
9  {
10     new OrderUpdate()
11     {
12         Placed = true,
13         Order = order
14     }
15 };

```

Executing a market order

The code for executing a buy market order is listed below. Executing a sell limit order is similar. The main loop runs while the market order is not fully matched and there are still distinct prices in the sorted list of ask prices. We can see this loop will run n times in the worst case where n is the number of distinct prices on the ask side of the order book. The next nested loop runs through each distinct price. There is another inner while loop which runs through orders at the current distinct price, matching quantity to the market order until there are no more orders or the market order is fully matched. This loop will run m times where m is the number of orders at the current price. The *Peek* function returns the oldest order in the queue while the *Dequeue* function removes the oldest order from the queue. Both these operations are $O(1)$. We can see that the complexity of this operation is $O(\sum m_i)$ in the worst case where m_i is the number of orders at distinct price i . This translates to all the orders on the ask side of the order book.

```

1  var quantity = order.Quantity;
2
3  var matches = new List<Match>();
4  var orderUpdates = new List<OrderUpdate>();
5
6  while (quantity > 0 && Asks.Count != 0)
7  {
8      var prices = Asks.Keys;
9      for (int i = 0; i < prices.Count; i++)

```

```

10     {
11         var price = prices[i];
12         var orders = Asks[price];
13
14         while (orders.Any() && quantity > 0)
15         {
16             var nextOrder = orders.Peek();
17
18             if (nextOrder.Quantity > quantity)
19             {
20                 nextOrder.Quantity -= quantity;
21                 matches.Add(new Match() { Price = price,
22                                     Quantity = quantity });
23                 orderUpdates.Add(new OrderUpdate()
24                 {
25                     Order = nextOrder,
26                     Matches = new List<Match> { new Match()
27                                             { Price = price, Quantity =
28                                             quantity} }
29                 });
30                 quantity = 0;
31             }
32
33             if (nextOrder.Quantity <= quantity)
34             {
35                 quantity -= nextOrder.Quantity;
36                 matches.Add(new Match() { Price = price,
37                                     Quantity = nextOrder.Quantity });
38                 orderUpdates.Add(new OrderUpdate()
39                 {
40                     Order = nextOrder,
41                     Matches = new List<Match> { new Match()
42                                             { Price = price, Quantity =
43                                             nextOrder.Quantity } }
44                 });
45                 orders.Dequeue();
46             }
47
48             if (!orders.Any()) Asks.Remove(price);
49         }
50     }
51 }
52
53 orderUpdates.Add(new OrderUpdate()
54 {
55     Placed = true,
56     Order = order,
57     Matches = matches
58 });

```

```
54
55 return orderUpdates;
```

Implementing force or kill validity

Section 2.3.2 describes force or kill validity as making sure a market order is entirely filled or not executed at all. Instead of partially executing an order and rolling back the order book, we first check if the sum quantity of all unmatched orders is greater than or equal to the quantity of the desired market order. We iterate through all limit orders to get this sum making this check $O(m)$ always. Here, m is the number of orders on the ask side of the order book for buy market orders or on the bid side for sell market orders. The code for buy market orders is shown below.

```
1  if (order.ExecutionValidity ==
    OrderExecutionValidity.ForceOrKill)
2  {
3      if (order.Quantity > Asks.Sum(a =>
        a.Value.Sum(l => l.Quantity)))
4      {
5          return new[] {new OrderUpdate()
6              {
7                  Placed = false,
8                  Order = order,
9                  Message = "FOK validity and
        not enough depth"
10             }};
11     }
12 }
```

4.4 Testing

Unit testing is an approach where your program is tested piece by piece (or unit by unit) in an automated fashion. As explained in Cheon and Leavens (2002), tests are written before, during and after code is written. These tests make sure of code correctness after writing as well as providing a base for regression testing. According to Schach (2004), regression tests ensure correctness of software after fixing or extending that software. It especially helps where there is pressure for a fix or extension and where there is little or no documentation. To test the simulator, we followed the AAA pattern as described by Lavesson (2012):

1. Arrange — Set up resources required to test the intended work. This can include setting up components and data.
2. Act — Typically involves calling the method you wish to test. Here you perform the work you wish to test.
3. Assert — Assert that the result of the work performed is as intended.

4.4.1 Test cases

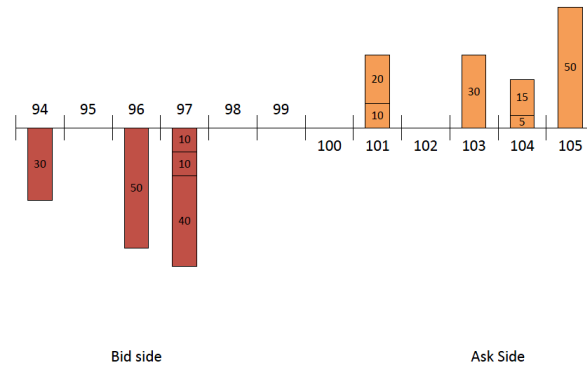
The limit order book mechanisms were verified using the scheme described in Section 4.4.

1. Arrange the limit order book into an initial state.
2. Act by submitting an order instruction.
3. Assert that the new order book state is what we expect.

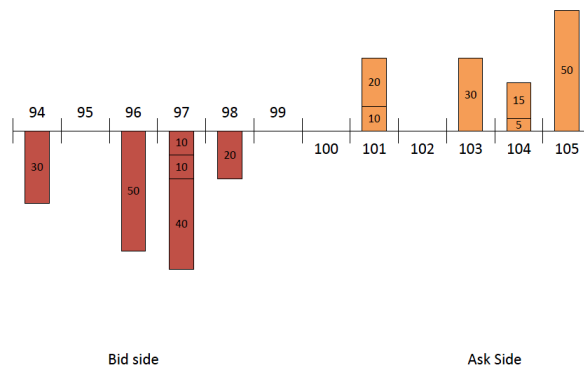
See Section B.1 for an code sample of a unit test.

Submit buy limit order at new price

1. Arrange

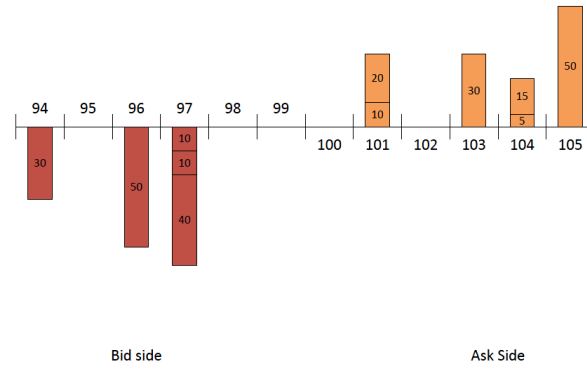


2. Act
Submit a buy limit order at price 98 and quantity 20.
3. Assert



Submit buy limit order at existing price

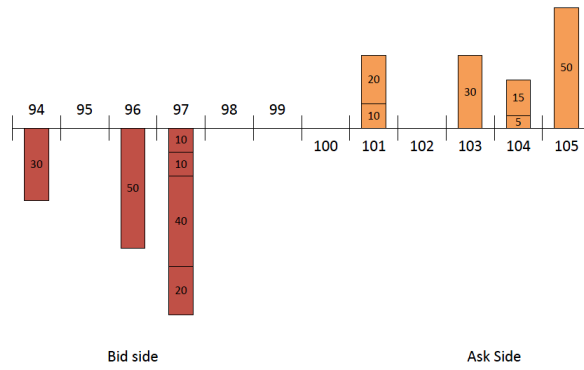
1. Arrange



2. Act

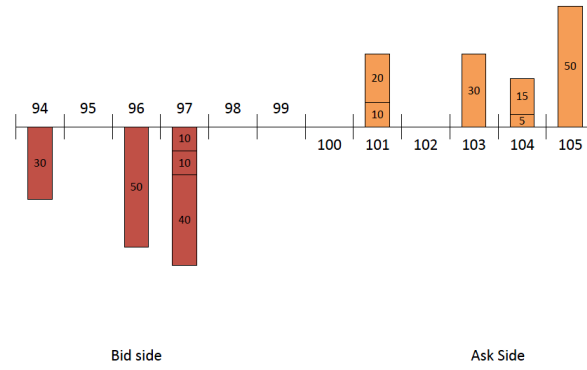
Submit a buy limit order at price 97 and quantity 20.

3. Assert



Submit sell limit order at new price

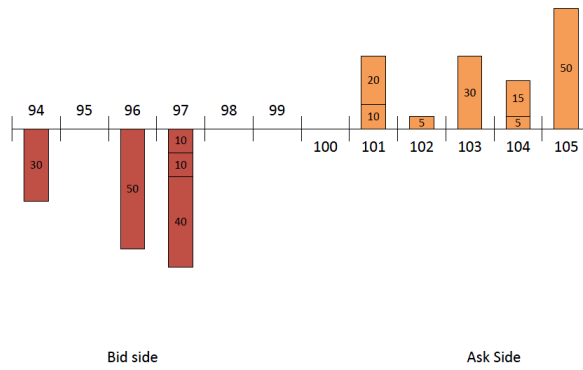
1. Arrange



2. Act

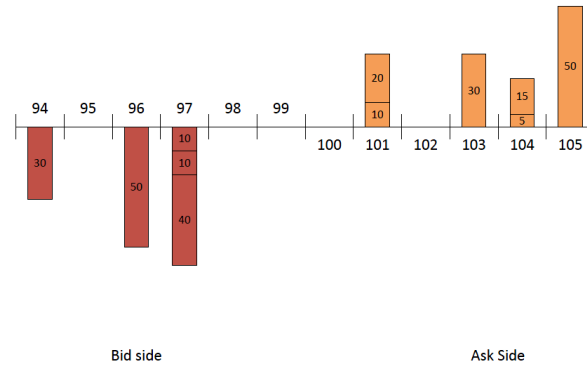
Submit a sell limit order at price 102 and quantity 5.

3. Assert



Submit sell limit order at existing price

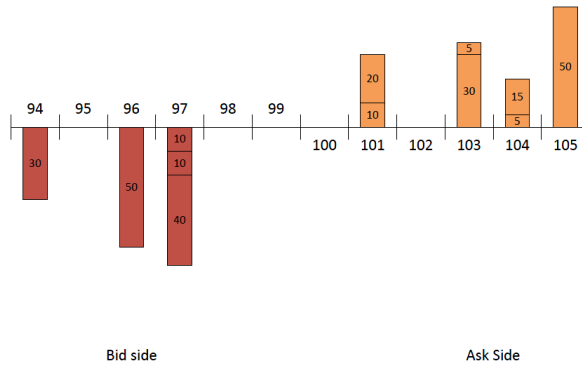
1. Arrange



2. Act

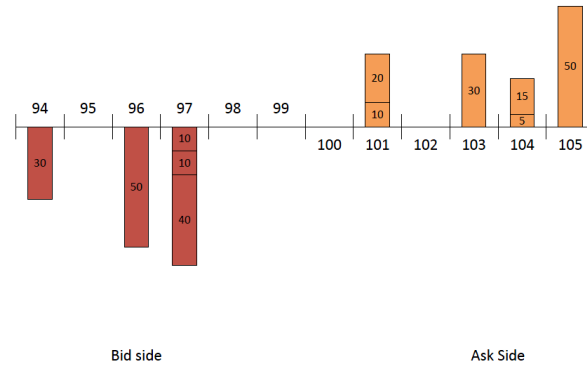
Submit a sell limit order at price 103 and quantity 5.

3. Assert



Submit buy market order

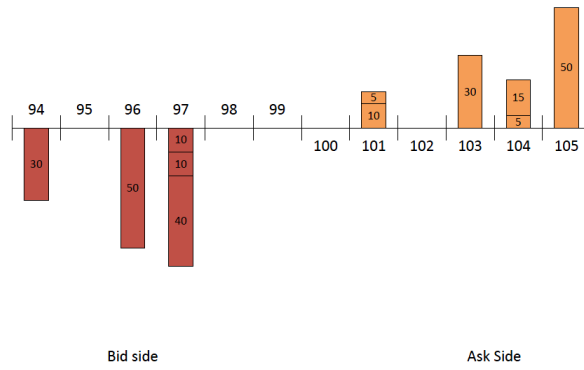
1. Arrange



2. Act

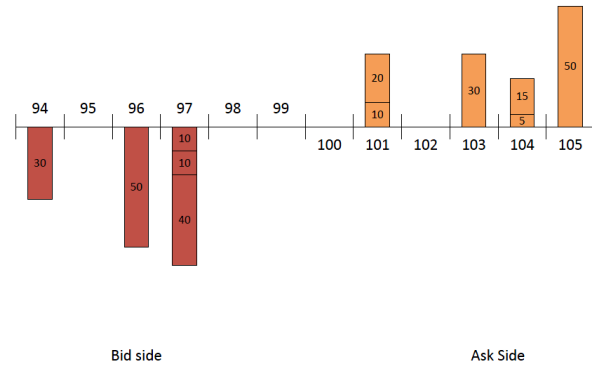
Submit buy market order at quantity 15.

3. Assert



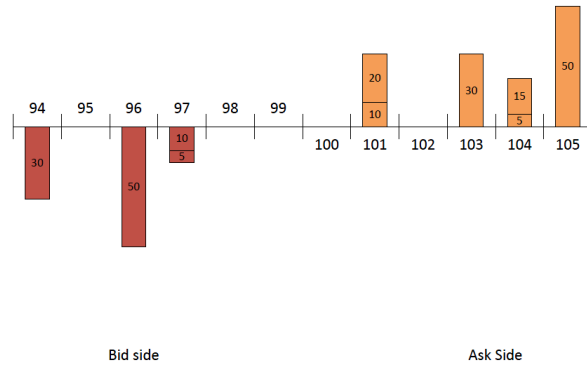
Submit sell market order

1. Arrange



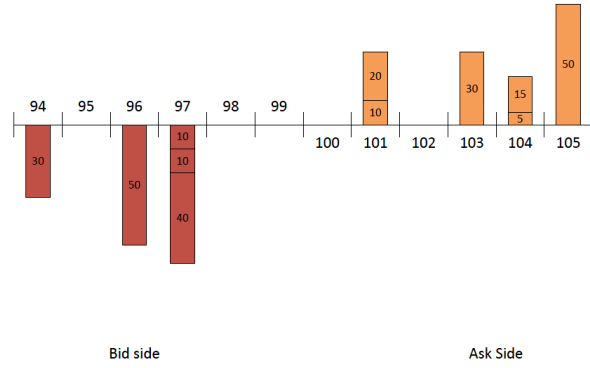
2. Act
Submit sell market order at quantity 45.

3. Assert



Submit buy market order which changes market depth

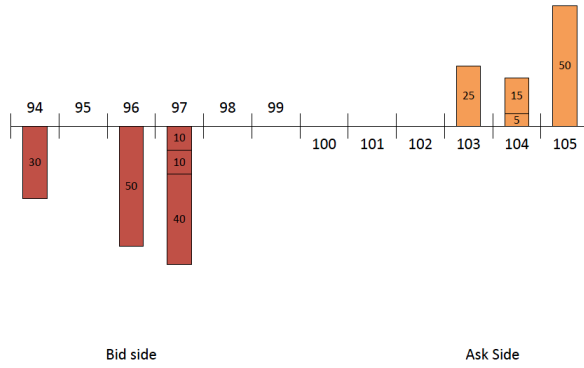
1. Arrange



2. Act

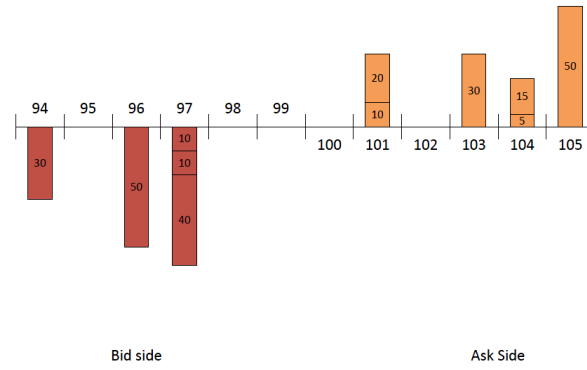
Submit buy market order at quantity 35.

3. Assert



Submit sell market order which changes market depth

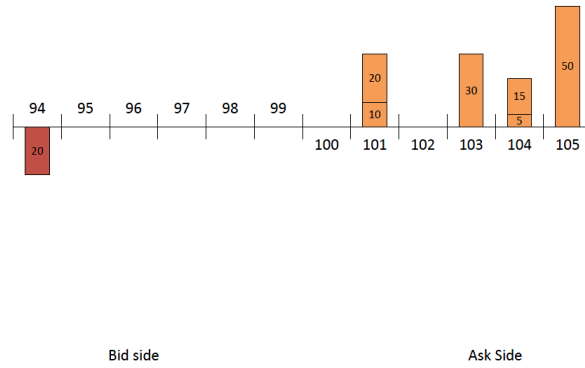
1. Arrange



2. Act

Submit sell market order at quantity 120.

3. Assert



Chapter 5

Showcase

In this chapter, we demonstrate applicability of the simulator by implementing agent based market models. We implement three models each exhibiting three types of simple interaction while yielding different realisations of price formation. Chiarella et al. (2009) allows us to showcase the ability of agents to react to the current external state of the market using their own internal decision rules. See Section 5.1 for the implementation. Gode and Sunder (1993) introduced the concept of Zero Intelligence Agents where agents make decisions based on their own randomized logic. In Section 5.2 we implement a Zero Intelligence Agent model. Here, agents make order decisions randomly without considering external factors. In Section 5.3 we implement the model described in Kluger and McBride (2011). Here, agents use learning mechanisms to fine tune their order decisions. We showcase the capability of agent interaction and social decision making using a genetic learning algorithm. We also show that agents are long running and can store their own past decisions in order to implement individual learning.

5.1 The impact of heterogeneous trading rules

In order to showcase the functionality of the market simulation framework, we first implement the model described in Chiarella et al. (2009). In this model, agents place market and limit orders based on utility maximization and future returns are forecast using a mixture of fundamentalist, chartist and noise induced components.

Pilbeam (1995) distinguishes fundamentalist traders as those who believe that asset price movements can be predicted using prospective changes in economic fundamentals rather than history. They assume perfect foresight of the fundamental economic variables and use these as input into often complex models to predict asset price behaviour. He goes on to describe chartist traders as those who use past patterns in an asset price to predict its value. In order to predict the behaviour of an asset price, they need only the recent history of that

asset price. Some traders base their decisions on irrational and often unwarranted pseudo-signals such as popular stock tips (Shleifer and Summers, 1990). Chiarella et al. (2009) introduces a model where agents make decisions based on a random mixture of all these strategies. The later strategy is expressed within the model as noise. A brief description of the model is given and some results presented.

5.1.1 The model

At each time step, an agent i is randomly chosen. The agent then forecasts its expected return over its desired time horizon τ and decides what action to take. We track the spot price, p_t , of the asset at time t . In this section, we use a superscript i to denote a parameter of agent i . For example: where S denotes the size a stock portfolio, S^i denotes the size of a stock portfolio belonging to agent i .

Initialization

Before the simulation begins, a desired number of agents are initialized. Each agent is randomly assigned a stock portfolio of size S^i and a cash amount C^i where $S^i \in [0, 50]$ and $C^i \in [0, p_0^f \times 50]$. Weights (g_1^i , g_2^i and n^i) are randomly sampled from Laplace distributions with location, $\mu_1 = \mu_2 = \mu_n = 0$, and desired scales b_1 , b_2 and b_n . These weights are used to determine the relative impact of each behavioural component in agent i 's decision making. The fundamentalist component is weighted by g_1^i while the chartist and noise induced components are weighted by g_2^i and n^i respectively. Each agent is also assigned a desired time horizon, τ^i , given by Equation (5.1) and a relative risk aversion level, α^i , given by Equation (5.2). Here, τ and α represent the simulation's reference time horizon and risk aversion level respectively. The reference time horizon and risk aversion levels of all agents can be adjusted using these parameters. The implication is that agents favouring a fundamentalist approach will consider a longer time horizon and be more risk averse when compared to those favouring a chartist approach.

$$\tau^i = \tau \frac{1 + g_1^i}{1 + g_2^i} \quad (5.1)$$

$$\alpha^i = \alpha \frac{1 + g_1^i}{1 + g_2^i} \quad (5.2)$$

Chiarella et al. (2009) introduces the asset's fundamental value, p_t^f , into the model. The fundamental value is the path predicted to be taken by the asset's price through time. As described earlier, fundamentalist traders use economic fundamentals and complex models to predict asset price movements. Here, we assume the fundamental value follows a geometric Brownian motion (GBM) with initial price $p_0^f = 300$, drift $\mu = 0$ and variance $\sigma = 0.0001$. Samuelson

(1971) shows that this is a valid assumption by comparing price predictions using GBM with historic markets. Appendix A is based on an introduction from Glasserman (2004) and provides more information on simulating the fundamental value using GBM. All agents know of the asset's simulated fundamental value and assume the model predicts future price movements perfectly. While this assumption is made by fundamental value driven agents, it has been shown that these models are not perfect and markets often don't display rational reflections of fundamental values (Summers, 1986).

Forecasting the expected return

In Chiarella et al. (2009), returns are calculated in logarithmic form. If $r_{t,t+1}$ is the return when a price moves from p_t to p_{t+1} then $r_{t,t+1} = \ln(\frac{p_{t+1}}{p_t})$. Also $p_{t+1} = p_t \exp(r_{t,t+1})$. Logarithmic returns are symmetric and additive. This means we can consider the average log return of an asset. The expected return is composed of three terms, each weighted randomly for every agent:

- the fundamentalist component,

$$\frac{1}{\tau_f} \ln\left(\frac{p_t^f}{p_t}\right), \quad (5.3)$$

which represents the normalized log return over time horizon τ_f ,

- the chartist component \bar{r}_t^i — the average log return of all time steps over time horizon τ_f ,

$$\bar{r}_t^i = \frac{1}{\tau^i} \sum_{j=1}^{\tau^i} \ln\left(\frac{p_{t-j}}{p_{t-j-1}}\right) \quad (5.4)$$

and

- the noise induced component

$$\epsilon_t. \quad (5.5)$$

These components are then added together and normalised. The expected return for the agent's desired time horizon is given by Equation (5.6) with the expected price given by Equation (5.7). Here, we denote \hat{r} to be the predicted or estimated return where r would denote the actual return.

$$\hat{r}_{t,t+\tau^i}^i = \frac{1}{g_1^i + g_2^i + n^i} \left[g_1^i \frac{1}{\tau_f} \ln\left(\frac{p_t^f}{p_t}\right) + g_2^i \bar{r}_t^i + n^i \epsilon_t \right] \quad (5.6)$$

$$\hat{p}_{t+\tau^i}^i = p_t \exp(-\hat{r}_{t,t+\tau^i}^i \tau^i) \quad (5.7)$$

Deciding what action to take

Chiarella et al. (2009) asserts that an agent considers the trade-off between expected returns and expected risk when deciding its portfolio composition. From the CARA (Constant Absolute Risk Aversion) demand function

$$U(W_t^i, \alpha^i) = -e^{-\alpha^i W_t^i}, \quad (5.8)$$

they derive the demand function

$$\pi^i(p) = \frac{\ln\left(\frac{\hat{p}_{t+\tau^i}^i}{p}\right)}{\alpha^i V_t^i p}, \quad (5.9)$$

where $\pi^i(p)$ denotes the number of stock agent i wishes to hold at price level p and

$$V_t^i = \frac{1}{\tau^i} \sum_{j=1}^{\tau^i} [r_{t-j} - \bar{r}_t^i]^2. \quad (5.10)$$

Here, V_t^i is the measure of variance of past returns and acts as a measure of risk. We can see that the higher the risk, the lower the demand at price p . The initial suggestion of utility functions exhibiting constant absolute risk aversion was made by Pratt (1964).

We can now estimate the agent's comfort price level p^* at which the agent is satisfied with its current stock portfolio (S_t^i) by setting $\pi^i(p) = S_t^i$ and solving numerically for p^* in

$$\pi^i(p^*) = \frac{\ln\left(\frac{\hat{p}_{t+\tau^i}^i}{p^*}\right)}{\alpha^i V_t^i p^*} = S_t^i. \quad (5.11)$$

The agent then chooses a value p in the interval $[p_m, p_M]$ where $p_M = \hat{p}_{t+\tau^i}^i$ and we solve for p_m in

$$p_m(\pi^i(p_m) - S_t^i) = C_t^i, \quad (5.12)$$

where C_t^i denotes agent i 's cash position at time t .

The agent then decides which action to take using the rules described in Table 5.1. See Section B.2 for a code sample of an agent which implements this behaviour.

Range	Order type	Volume	Price
$p_m < p < a_t^q$	Buy limit order	$\pi^i(p) - S_t^i$	p
$a_t^q \leq p < p^*$	Buy market order	$\pi^i(a_t^q) - S_t^i$	a_t^q
$p = p^*$	No action		
$p^* < p \leq b_t^q$	Sell market order	$S_t^i - \pi^i(b_t^q)$	b_t^q
$b_t^q < p \leq p_M$	Sell limit order	$S_t^i - \pi^i(p)$	p

Table 5.1: Decision table used by the agent where a_t^q and b_t^q are the best quoted ask and bid prices respectively.

5.1.2 Implementation considerations

At each time step, a single agent is randomly selected. This allows for a simple implementation but leaves no room for performance increases using parallel techniques. Optimization effort can only be focused on the serial logic used by the agent to make its order decision. We also optimized the simulation by calculating p_t^f (the fundamental value) upfront and only once even though it is only needed for the selected time horizon at each step. When deciding what action to take, we calculate the average variance of past returns — V_t^i . When selecting a small time horizon, this value can be close to zero. This raises computational issues when calculating the comfort price in Equation 5.11. We have dealt with this situation by selecting large enough time horizons during the simulation. However, we still found that V_t^i is close to zero during periods of low volatility. In these situations, the agent chooses to take no action and a warning is logged to the console.

5.1.3 Results

A number of simulations were run with 5000 agents. The following parameters were kept constant: $S^i = 50$, $C^i = 15000$, $\tau = 200$ and $\alpha = 0.01$. The noise component is a random sequence of samples from a normal distribution with mean 0 and variance 0.0001. The values of b_1 , b_2 and b_n were varied. These parameters affect the size of the contribution of the fundamental, chartist and noise components respectively. We record p_t and p_t^f at each time step.

We first set $b_1 = 10$ and $b_2 = b_n = 0$. The result is plotted in Figure 5.1. Figure 5.2 plots the result where only the noise component is taken into account ($b_1 = b_2 = 0, b_n = 1$). We then incrementally set $b_1 = 10$ and $b_2 = 1.2$ in Figure 5.3 and Figure 5.4 respectively. Figure 5.5, Figure 5.6 and Figure 5.7 show snapshots of the limit order book at different stages of the final simulation.

5.1.4 Interpretation of results

The results shown in Section 5.1.3 closely resemble those from Chiarella et al. (2009). As expected, Figure 5.1 shows the price tracking the fundamental value when only the fundamentalist component is taken into account. When only the noise component is taken into account (Figure 5.2), a downward trend in price is displayed. This shows that agents would rather sell than hold when there is little variance between the current and expected price. A high variance will also cause agents to sell when the reference risk aversion level is set high. Figure 5.3 shows the price loosely following the trend of the fundamental price but which high volatility due to the noise component. Figure 5.4 introduces the chartist component into consideration. When compared with Figure 5.3, we see periods of clustered volatility where the variance in price is unpredictable. Figure 5.8 shows a comparison of the variance of the spot price where only the fundamental and noise components are considered with the variance of the spot price where

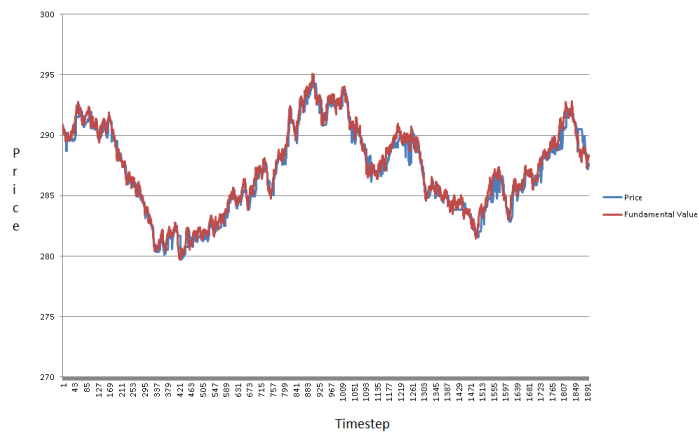


Figure 5.1: Time series data where all agents consider only their perfect information about the fundamental value of the simulated stock.

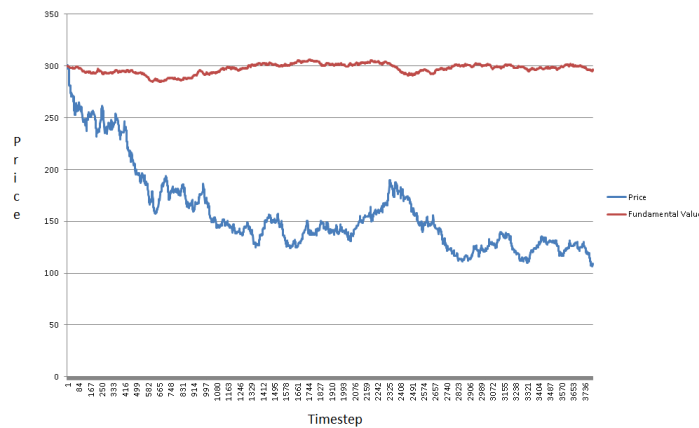


Figure 5.2: Time series data where only the noise component is taken into account. Agents do not consider the fundamental value.

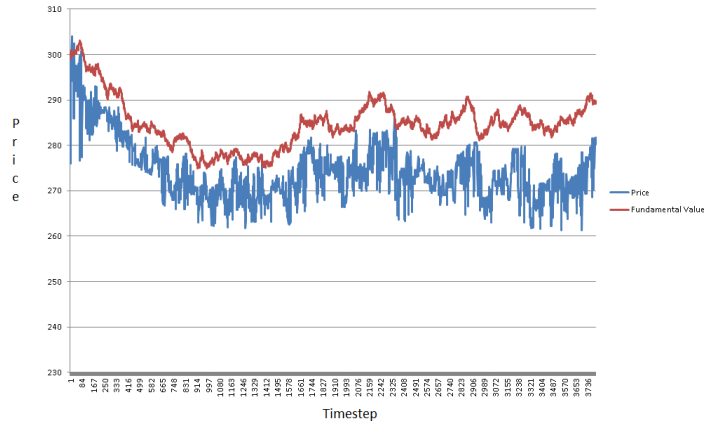


Figure 5.3: Time series data where the fundamental component is taken into account but agents do not believe this information is perfect and this noise effects their decision making.

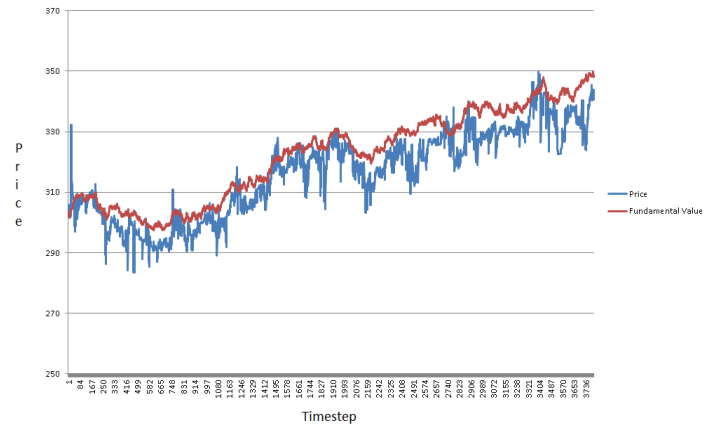


Figure 5.4: Time series data where both fundamental and chartist behaviour is taken into account as well as the effect of noise in decision making.



Figure 5.5: Limit order book snapshot at time step 2000 for final simulation.

all components are considered (Figure 5.3 compared to Figure 5.4). We use a sliding window of 10 time steps to show how variance changes over time. Here we see the variance when all components is larger and more clustered when compared to the variance when only the fundamental and noise components are in effect.



Figure 5.6: Limit order book snapshot at time step 3000 for final simulation.

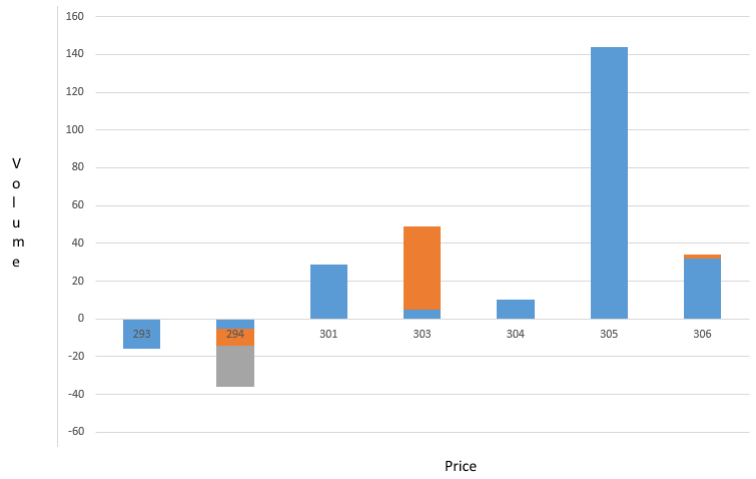


Figure 5.7: Limit order book snapshot at time step 4000 for final simulation.

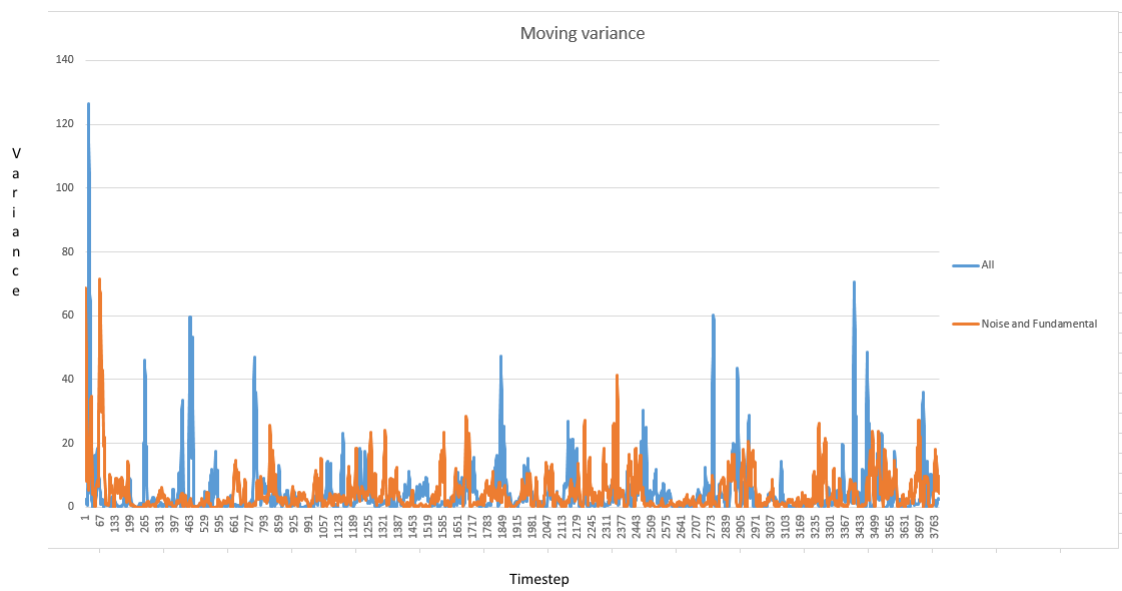


Figure 5.8: Comparison variance of spot price where only the fundamental and noise components are considered and where all components are considered over a sliding window of size 10.

5.2 Zero intelligence agents

Gode and Sunder (1993) introduced zero intelligence agents as agents which submit random bids and offers subject to some constraints. Farmer et al. (2004) goes on to use a zero intelligence model to explain stylized facts seen in data from the London Stock exchange. We introduce a simple zero intelligence agent model.

5.2.1 The model

In this model, agents use random number generators to decide on order types, quantity and price (for limit orders). The scheme works as follows:

1. Initialize n_{lm} liquidity makers with the parameters V_{max} , the maximum price variance, and Q_{max} , the maximum order size.
2. Initialize n_{lt} liquidity takers with the parameter Q_{max} as above.
3. Poll every agent in random order for their next action with the following probabilities: p_n — do nothing, p_b — submit a buy order, p_s — submit a sell order.
4. For liquidity takers, generate a random number, p on the range $[0, 1]$. If $p \leq p_n$, do nothing. If $p_n < p \leq p_n + p_b$, submit a buy market order. Otherwise, submit a sell market order. The quantity of the order is $q = q_n Q_{max}$ where q_n is normally distributed between 0 and 1.
5. The procedure for liquidity makers is similar. Generate a random number, p on the range $[0, 1]$. If $p \leq p_n$, do nothing. If $p_n < p \leq p_n + p_b$, submit a buy limit order. Otherwise, submit a sell limit order. The quantity of the order is $q = q_n Q_{max}$ where q_n is randomly sampled from a normal distribution in the range $[0, 1]$. The limit order price is $P = A_b - v_n V_{max}$ for buy limit orders and $P = O_b + v_n V_{max}$ for sell limit orders. A_b is the best ask price, O_b is the best offer price and v_n is randomly sampled from a normal distribution with mean 0 and standard deviation p_σ .

See Section B.2 for a code sample for the liquidity taker and Section B.3 for the liquidity maker.

5.2.2 Results

The simulation was run with $n_{lm} = n_{lt} = 200$, $V_{max} = 2$, $Q_{max} = 30$, $p_n = 0.8$ and $p_b = p_s = \frac{1-p_n}{2}$. These were kept constant while p_σ was set to 0.5, 2 and 10. In Figures 5.9 to 5.17 we plot the bid and ask, spread and log return time series to illustrate the effect of changing p_σ .

As we increase p_σ , the prices agents choose show more volatility. This is seen in the increase in the standard deviation of the mid price. As volatility increases, so does the bid-ask spread. Table 5.2 summarizes the results.

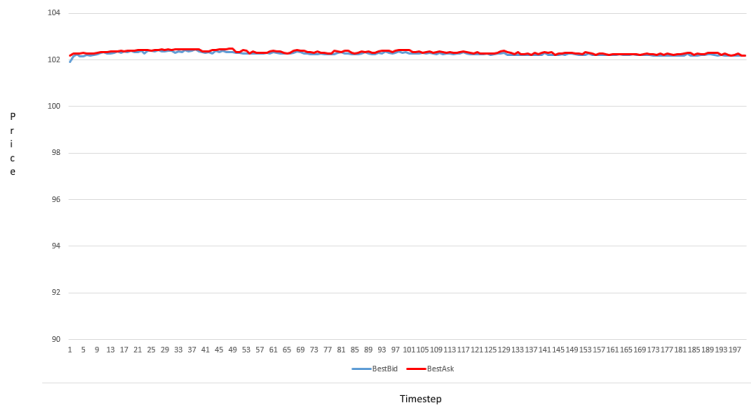


Figure 5.9: Best bid and ask time series with $p_\sigma = 0.5$.

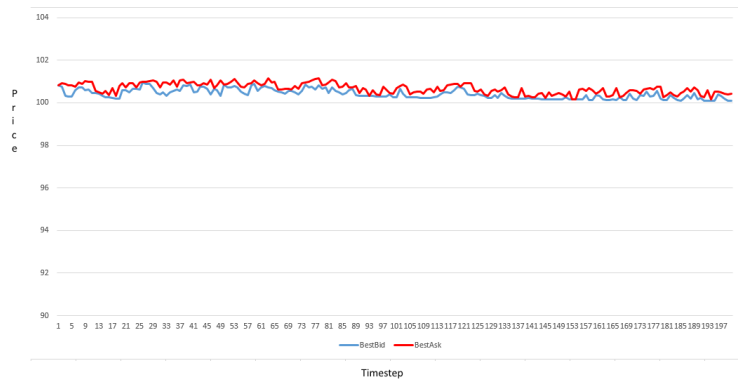


Figure 5.10: Best bid and ask time series with $p_\sigma = 2$.

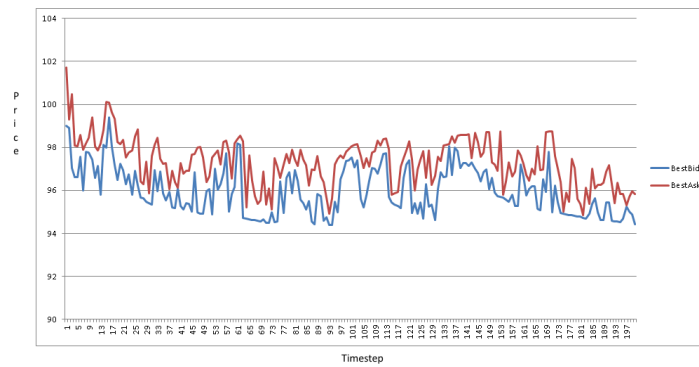


Figure 5.11: Best bid and ask time series with $p_\sigma = 10$.

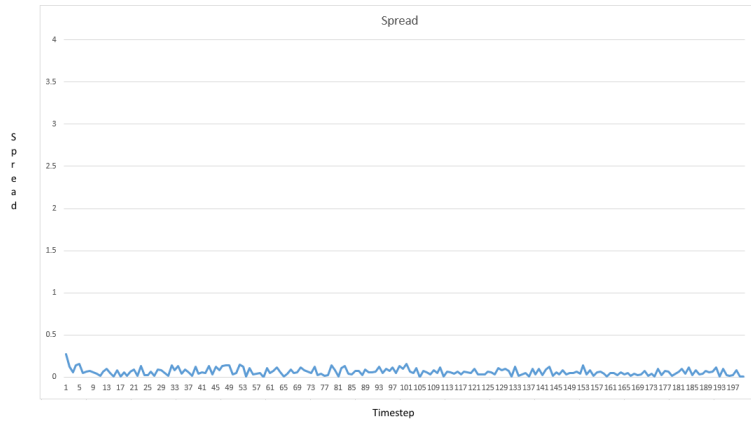


Figure 5.12: Spread time series with $p_\sigma = 0.5$.

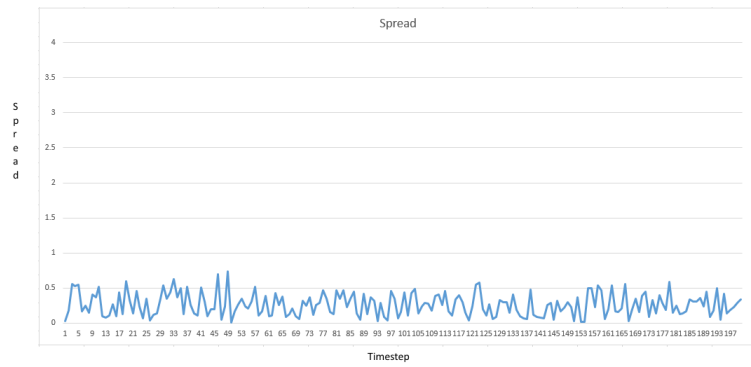


Figure 5.13: Spread time series with $p_\sigma = 2$.

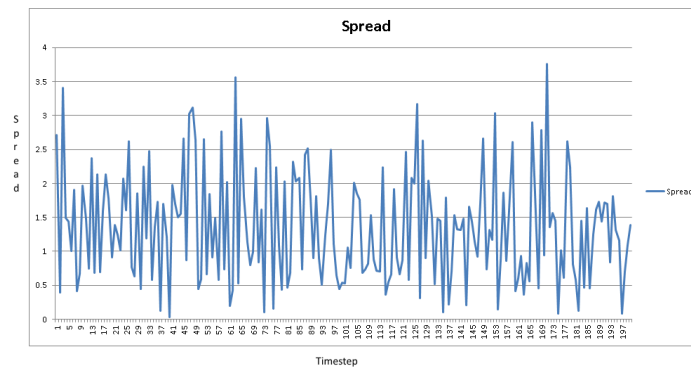


Figure 5.14: Spread time series with $p_\sigma = 10$.

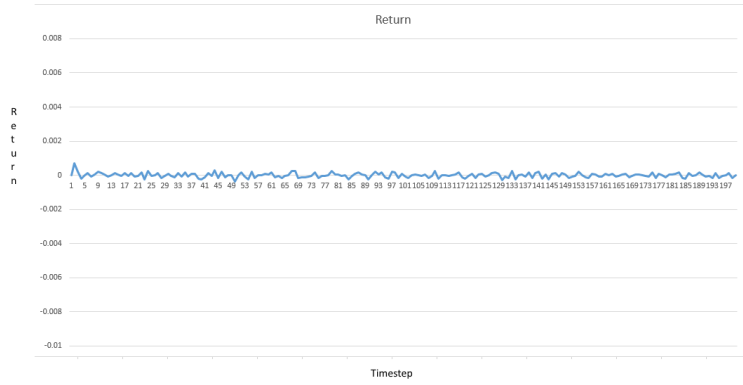


Figure 5.15: Return series with $p_\sigma = 0.5$.

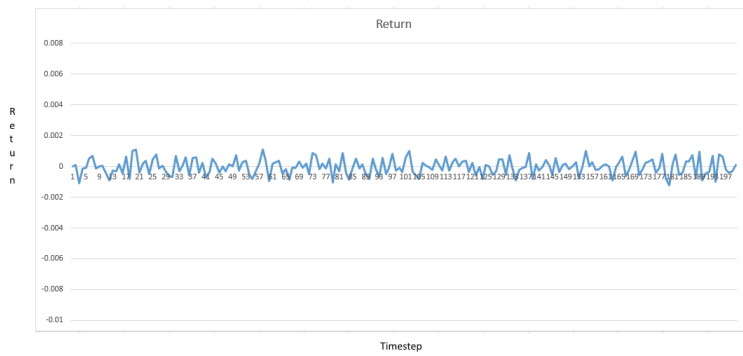


Figure 5.16: Return time series with $p_\sigma = 2$.

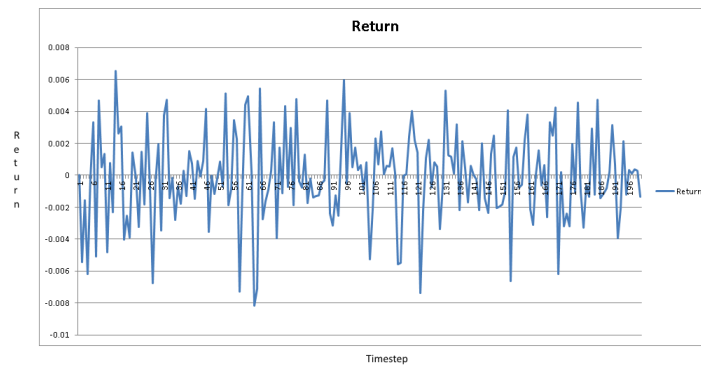


Figure 5.17: Return time series with $p_\sigma = 10$.

	Mid Price		Spread		Return	
p_σ	Mean	Std Dev	Mean	Std Dev	Mean	Std Dev
0.5	102.292296	0.065681331	0.060849	0.04112598	0.00000030154	0.000138353
2.0	100.5415968	0.220053679	0.2617805	0.160866188	-0.00000122738	0.00050748
10.0	96.64768175	1.02790073	1.3619665	0.814826612	-0.000116176	0.002807781

Table 5.2: Results for zero intelligence agent model.

5.2.3 Interpretation of results

This model is relatively simple: agents' decisions are random and their price point is selected looking only at the current best bid and ask prices. Unsurprisingly, we see little emergent behaviour. We do see that the variability of log returns increase with an increase in p_σ . The increase in this volatility is also linked to an increase in the bid-ask spread. However, this volatility is fairly uniform and shows little resemblance to the volatility clustering shown in Qi (2009). While this model can be used as a good base, it does not admit many realistic market characteristics. In Section 5.3 we show a model based on zero intelligence agents which includes social and individual learning.

5.3 Agent learning and intra-day trading patterns

Kluger and McBride (2011) introduces an agent-based stock market where agents are essentially zero intelligence agents but have the additional capability of learning. Agents learn through social interaction or analysing the result of their own individual behaviour. There are two groups of agents:

- informed agents who have information regarding the exact asset pricing during the current period, and
- uninformed agents who form their own information on what the asset price should be during the current period.

Informed agents can trade during any (or no) period throughout the day. Uninformed agents must trade during only one period throughout each day. We have implemented their model on our platform and extracted results to study emergent behaviour. The code for the model is documented in Appendix B.5.

5.3.1 The model

In this section, we describe the model introduced in Kluger and McBride (2011). Table 5.3 provides a list of the parameters we refer to as well as their initial values.

Parameter	Description	Base value
N	Number of agents	200
D	Number of days	2250
T	Number of trading periods per day	8
I	Percentage of informed agents	20
A_d	Asset value on day d	$U[20, 120]$
E_{min}, E_{max}	Range of expected asset value	$U[20, 120]$
R_{min}, R_{max}	Range around an agents expected asset value in which it will trade	$U[5, 15]$
\bar{L}	Number of days in a learning period	15
Informed agents learn		True
Uninformed agents learn		True
Informed agents compete	Informed agents compete by also placing limit orders	False
G_s	Size of groups in genetic algorithm	40
G_{cp}	Number of crossover points in genetic algorithm	4
G_m	Mutation probability in genetic algorithm	0.001
β	Boltzman temperature cooling in individual learning	30
η	Experimentation in individual learning	0.15
$q_j(0)$	Initial propensity in individual learning	8000
θ	Recency in individual learning	0.1

Table 5.3: Model parameters and base values.

Condition	Order side	Order type
$r_i < E_{id}$ and $r_i < bestask$	Buy	Limit
$r_i < E_{id}$ and $r_i \geq bestask$	Buy	Market
$r_i > E_{id}$ and $r_i > bestbid$	Sell	Limit
$r_i > E_{id}$ and $r_i \leq bestbid$	Sell	Market

Table 5.4: Agent decision based on r_i , E_{id} , the current best bid and the current best ask.

Initialization and timing chromosomes

We initialize N agents of which $I\%$ are informed agents and $(100 - I)\%$ are uninformed agents. Each agent is assigned a random *timing chromosome*. An agent bases its decision on whether to trade in a certain period on its timing chromosome. For uninformed agents, their timing chromosomes are represented as 3 bit binary numbers. Each bit combination represents the period the agent will enter the market. For example: an agent with timing chromosome 000 will enter during period 0, with 001 during period 1, etc. Informed agents have an 8 bit binary number where the value of the bit in position a (starting from the right) represents that agent going to the market during period a . For example: an agent with timing chromosome 00000000 will not go to the market at all. An agent with timing chromosome 00000001 will go to the market during period 0 and an agent with timing chromosome 11111111 will trade in the market during all periods (0, 1, 2, 3, 4, 5, 6 and 7).

Choosing an expected asset value

At the beginning of each day we choose an actual asset value, A_d , for simulation. A_d is randomly selected from the range $[E_{min}, E_{max}]$. Informed agents are called such because they know what the actual asset value is. For them, E_{id} (agent i 's expected asset value) is set to A_d . Uninformed agents form their own expectation of the asset value. They choose a value between E_{min} and E_{max} as their expected asset value E_{id} .

Deciding on an action

During each period, an agent uses its timing chromosome to decide on whether it will trade. If it will trade, it decides on what type of trade to put in. The decision making process is similar to that used by the zero intelligence agents in Section 5.2. An agent chooses a random value, r_i , between $E_{id} - R_i$ and $E_{id} + R_i$. Here, R_i is chosen randomly between R_{min} and R_{max} . Table 5.4 summarizes the decision agent i will take. The profit (or reward) associated with this action is the difference between r_i and A_d .

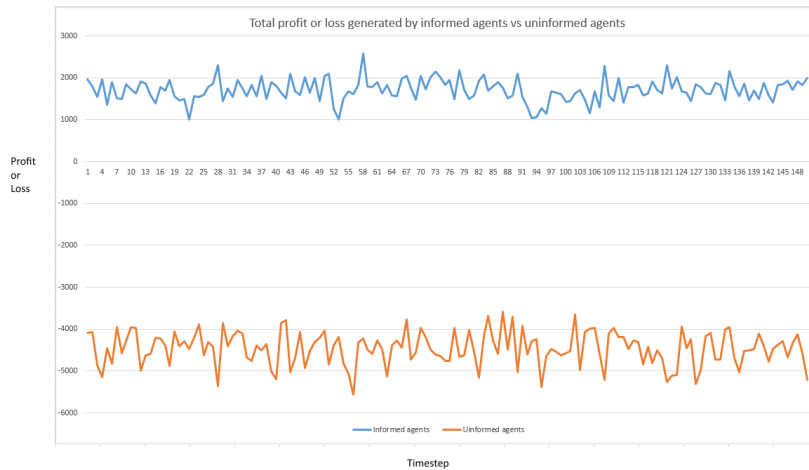


Figure 5.18: Total profits generated by informed and uninformed agents under social learning.

Learning

A learning period is \bar{L} days long. At the end of each learning period, all agents adjust their timing chromosomes using either social learning (a genetic algorithm) or individual learning (modified Roth-Erev learning). During social learning, the genetic algorithm is run in three phases.

1. Selection: Uninformed agents are split into groups while informed agents form their own group. Each agent then selects another agent's timing chromosome to form a pair. The likelihood of a timing chromosome being selected for pairing is determined by the reward agents which chose it earned during the learning period. As mentioned, the reward is equal to the profit (or loss) made when this timing chromosome is used and is the difference between r_i and A_d . As an example, Figure 5.18 plots the total profit generated by informed and uninformed agents during each period under social learning.
2. Crossover: A crossover point is randomly selected where the timing chromosomes will swap bits.
3. Mutation: The new timing chromosome then goes through a mutation process where each bit has a small chance of mutating (flipping).

Social learning requires an agent to have a full view of all other agents' timing strategy and the reward associated with each strategy. The more rewarding timing chromosomes become the most selected. Over time, agent coordination occurs due to the dominance of a few profitable timing chromosomes.

During individual learning, an agent knows only its own past actions and associated rewards. Kluger and McBride (2011) deploys a modified Roth-Erev learning mechanism.

- After each learning period, an agent chooses from a list of all possible timing chromosomes. Each timing chromosome has its own propensity at time $t - q_j(t)$.
- The probability of a timing chromosome being selected is $Prob_t(j) = \frac{e^{q_j(t)/\beta}}{\sum_{n=1}^N e^{q_n(t)/\beta}}$ where β is the temperature (*cooling*) parameter.
- All timing chromosomes start the beginning of the simulation with the same propensity — $q_j(0)$.
- After each learning period, each agent updates all the propensities for each timing chromosome according to $q_j(t+1) = [1 - \theta]q_j(t) + E_j(\eta, N, k, t)$, where k is the last chosen action, $r_k(t)$ is the profit earned by choosing k and $E_j(\eta, N, k, t) = \begin{cases} r_k(t)[1 - \eta], & j = k \\ q_j(t)\frac{\eta}{N-1}, & j \neq k \end{cases}$

5.3.2 Results and interpretation

The simulation was run using the parameters listed in Table 5.3 using different approaches for timing chromosome selected: the genetic algorithm (social learning), individual modified Roth-Erev learning and random selection. Under random selection, an agent is assigned a random timing chromosome. We observe the total volume traded in each learning period as well as a measure for coordination. The coordination index is defined by $\sum_{j=1}^T s_j^2$ where s_j is the fraction of agents processing chromosome j . The results obtained were in line with those in Kluger and McBride (2011). We discuss the results obtained from each approach.

Social learning

Under the social learning scenario, it was noted that agents learn to trade early in the day. Figure 5.19 shows a drop in trading volume emphasized in the second half of the trading day. Social learning also leads to a high amount of coordination between agents. As shown in Figure 5.20, uninformed agents quickly coordinate and reach peak coordination levels between the 90th and 100th generations. Informed agents show a steady and consistent increase in coordination until the 50th generation where a slight drop and flattening in the coordination index occurs.

Emergent patterns

The results obtained show that, when agents are subject to learning, trade volume is higher during earlier periods of the day. Under random selection,

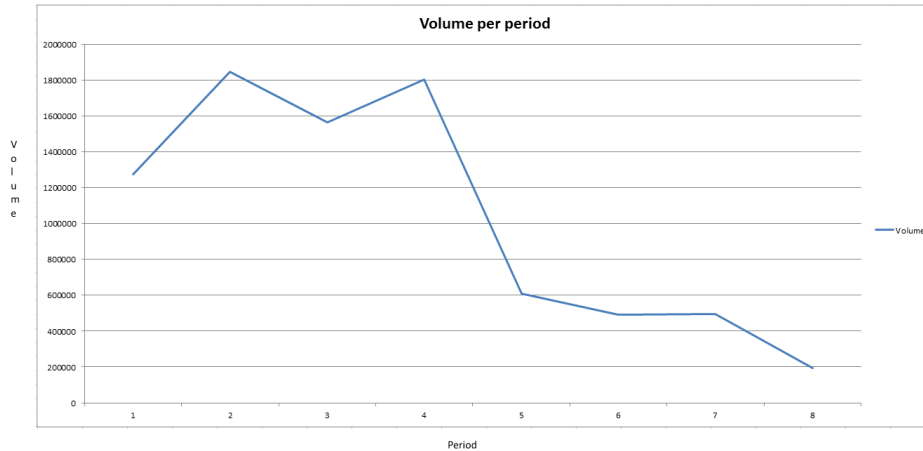


Figure 5.19: Volume traded per trading session under social learning.

no learning takes place and trading volume is flat throughout the entire day. The dominance of timing chromosomes which dictate early intra-day trading is linked to their high profitability when compared with timing chromosomes which dictate trading later in the day. As the order book is cleared at the end of each trading day, this higher profitability can be linked to limit orders placed earlier in the day having a greater chance of being matched when compared to those placed later in the day.

Individual learning

Similar to social learning, individual learning also produces a decline in trading volume as the trading day progresses. This is clearly seen in Figure 5.21. The coordination index plotted in Figure 5.22 is very low and stable. This is consistent with our model assumptions: during individual learning, agents have no information on other agents' trading patterns or strategies - only their own.

Random selection

Random selection produces relatively flat trading volume throughout the trading day with volume ranging between 986345 and 1129372 (Figure 5.23). In contrast, individual learning shows volume starting at 1865482 in the first period and going down to 108342 in the last. As expected, coordination levels show no clear pattern in Figure 5.24. Uninformed agents show a larger but unstable level of coordination but this is due to the relatively low number of possible timing chromosomes (8 vs 256 for informed agents). The level of coordination is very low for informed agents showing the random nature of chromosome selection.

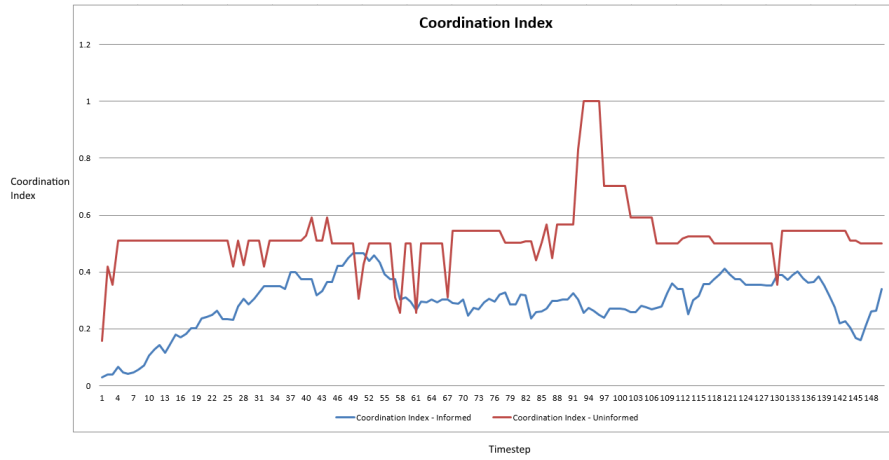


Figure 5.20: Coordination index per generation under social learning.

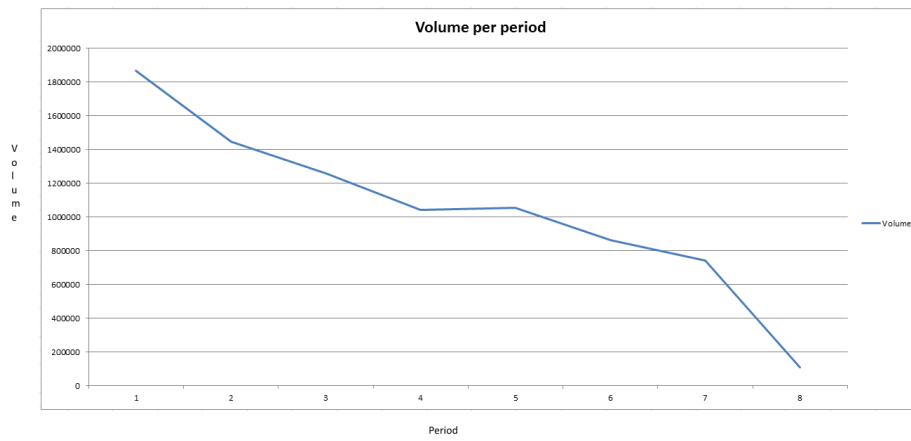


Figure 5.21: Volume traded per trading session under individual learning.

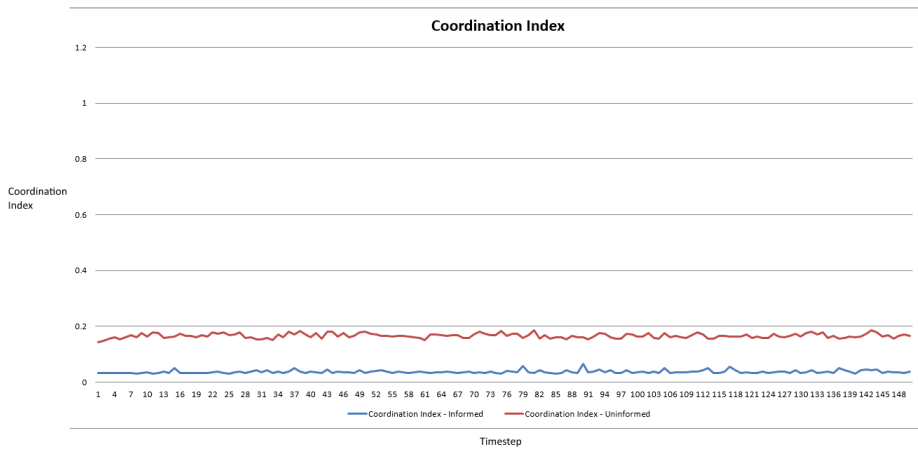


Figure 5.22: Coordination index per generation under individual learning.

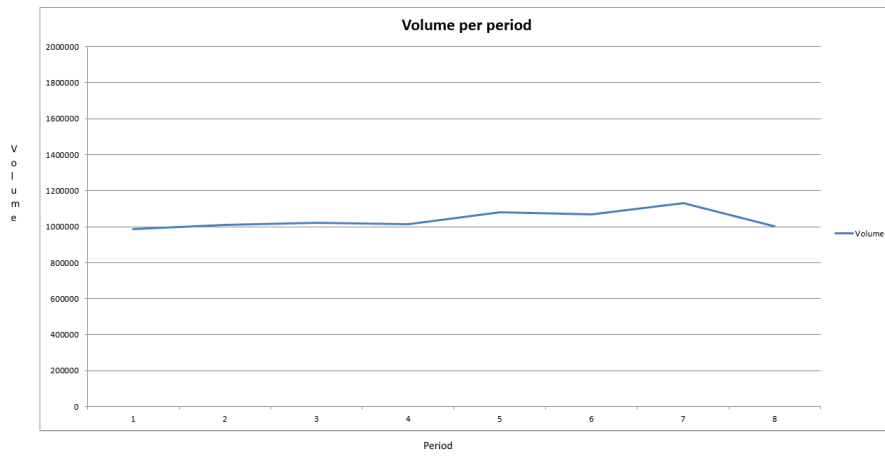


Figure 5.23: Volume traded per trading session under random selection.

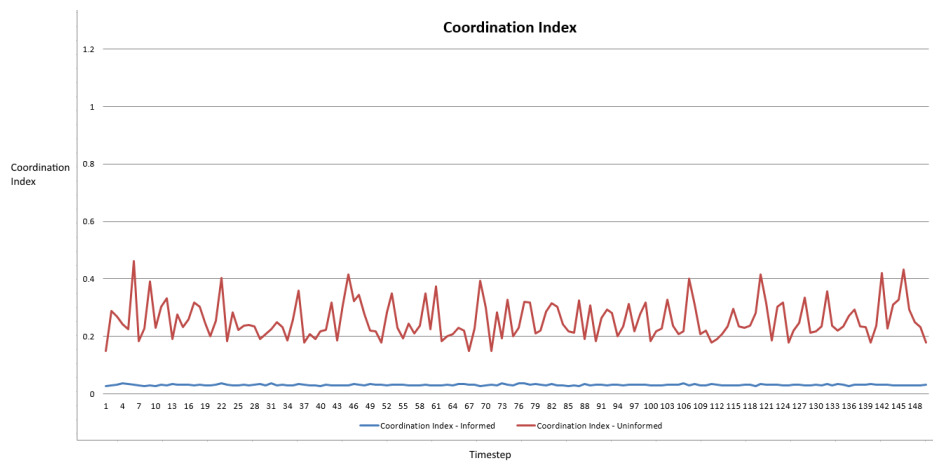


Figure 5.24: Coordination index per generation under random selection.

Chapter 6

Discussion

6.1 Conclusion

The focus of this work has been the design and creation of an agent based simulator. The ability to simulate markets using agent based models has also been illustrated. We have shown the specification, design, implementation and verification of an agent based single asset market simulator based on manuals obtained from the JSE. The entire process has been documented and presented in this thesis — something often overlooked in current literature. Three agent based models were implemented to showcase the use of the simulator. Complex agents were introduced into the environment where simulations were run and price behaviour tracked. The ability to measure the output of these agent models and, further, the ability to use these outputs to verify the correctness of these models against real data was shown. In its current state, the simulator does have limitations but is designed to allow for further extension.

6.2 Limitations

Dark pools and auction periods were out of scope for this work. All other order types were specified and included in the design of the simulator. However, not all of these have been implemented and verified. While agents are able to base decisions on any factors, the focus of this simulator remains on a single asset market.

6.3 Further work

Completing the implementation of all the order types and including hidden orders and dark pool functionality will give the user a tool which is closer to reality. The technical design if the simulator takes into account future expansion and relies on interfaces to ensure components are separated neatly and can be easily

changed. These changes can then be tested using the verification techniques discussed. There is also further opportunity outside of extension of the simulator: the simulator allows the user to focus on the development of different agent behaviour while reusing a pre-built, flexible, verified simulation framework. This opens opportunities to develop and run agent models to show stylized market behaviour or to test trading strategies. Considering implementation, there is also scope for the design of the simulator to be reused and implemented using different programming languages and frameworks.

Appendices

Appendix A

Geometric Brownian Motion in C#

This section explains concepts set out in Glasserman (2004) on Geometric Brownian Motion. We also include C# code which simulates the path of a stock price using a discretized scheme.

A.1 Background

Samuelson (1971) shows that a model for price speculation based on Brownian motion shows quantitative properties which resemble those of historical markets.

Glasserman (2004) defines a standard one-dimensional Brownian motion, $W(t)$, and further states that a process $X(t)$ is a Brownian motion with drift μ and diffusion coefficient σ^2 if

$$\frac{X(t) - \mu t}{\sigma} \tag{A.1}$$

is a standard Brownian motion. Solving for X , we see

$$X(t) = \mu t + \sigma W(t). \tag{A.2}$$

X is then the solution to the stochastic differential equation

$$dX(t) = \mu dt + \sigma dW(t). \tag{A.3}$$

A geometric brownian motion, $S(t)$, is an exponentiated Brownian motion iff $\log S(t)$ is a brownian motion with initial value $\log S(0)$. Glasserman (2004) shows that if S is a Geometric Brownian process, then

$$S(t) = S(0)e^{[\mu - \frac{1}{2}\sigma^2]t + \sigma W(t)}. \tag{A.4}$$

Given the initial value $S(0)$, we can solve for the value at any time t . This leads to a recursive procedure for any time t_i where $t_0 < t_i < t_n$:

$$S(t_{i+1}) = S(t_i)e^{[\mu - \frac{1}{2}\sigma^2](t_{i+1} - t_i) + \sigma\sqrt{t_{i+1} - t_i}Z_{i+1}}, \tag{A.5}$$

where Z_1, Z_2, \dots, Z_n are independent standard normals. Appendix A.2 shows C# code which follows the discretization in Equation (A.5) to generate the random walk of a stock price.

A.2 Code

The C# implementation of the scheme shown in Equation A.5 follows. For the generation of independent random normal variables, we use the Math.NET Numerics library (MathDotNet).

```

1 private static double[] GenerateFundamentalValuePath(int
    simulationSteps, double fundamentalValueInitial, double
    fundamentalValueDrift, double fundamentalValueVariance)
2 {
3     Normal standardNormal = new Normal(0,
        fundamentalValueVariance);
4
5     var fundamentalValue = new
        double[simulationSteps];
6
7     fundamentalValue[0] =
        fundamentalValueInitial;
8
9     for (int i = 1; i < simulationSteps; i++)
10    {
11        fundamentalValue[i] =
            CalculateNextValue(1,
                fundamentalValue[i-1],
                0,
                fundamentalValueVariance,
                standardNormal);
12    }
13
14    return fundamentalValue;
15 }
16
17 private static double CalculateNextValue(double timeStep,
    double currentValue, double drift, double variance,
    Normal normalDistribution)
18 {
19     return currentValue *
        Math.Exp(((drift-0.5)*Math.Pow(variance,2)*timeStep)
        +
        variance*Math.Sqrt(timeStep)*normalDistribution.Sample());
20 }

```

Appendix B

Code Samples

This section includes code samples referred to in previous sections. The full source code can be found at <https://github.com/preyen/marketsimulator>.

B.1 Example Test case

```
1 [TestMethod]
2 public void CanSubmitBuyMarketOrder15()
3 {
4     //ARRANGE
5     ILimitOrderBook lob = new
6         LimitOrderBook.StandardLimitOrderBook();
7
8     Assert.IsNotNull(lob);
9     Assert.AreEqual(0, lob.Bids.Count);
10
11     Assert.IsInstanceOfType(lob, typeof(ILimitOrderBook));
12
13     var setupOrder = new Order() { Price = 101, Quantity =
14         20, Side = OrderSide.Sell, Type =
15         OrderType.LimitOrder, UserID = "Test" };
16     var result = lob.ProcessLimitOrder(setupOrder);
17     Assert.AreEqual(1, result.Count());
18     Assert.AreEqual(true, result.First().Placed);
19
20     setupOrder = new Order() { Price = 101, Quantity = 10,
21         Side = OrderSide.Sell, Type = OrderType.LimitOrder,
22         UserID = "Test" };
23     result = lob.ProcessLimitOrder(setupOrder);
24     Assert.AreEqual(1, result.Count());
25     Assert.AreEqual(true, result.First().Placed);
26 }
```

```

22     setupOrder = new Order() { Price = 103, Quantity = 30,
        Side = OrderSide.Sell, Type = OrderType.LimitOrder,
        UserID = "Test" };
23     result = lob.ProcessLimitOrder(setupOrder);
24     Assert.AreEqual(1, result.Count());
25     Assert.AreEqual(true, result.First().Placed);
26
27     setupOrder = new Order() { Price = 104, Quantity = 15,
        Side = OrderSide.Sell, Type = OrderType.LimitOrder,
        UserID = "Test" };
28     result = lob.ProcessLimitOrder(setupOrder);
29     Assert.AreEqual(1, result.Count());
30     Assert.AreEqual(true, result.First().Placed);
31
32     setupOrder = new Order() { Price = 104, Quantity = 5,
        Side = OrderSide.Sell, Type = OrderType.LimitOrder,
        UserID = "Test" };
33     result = lob.ProcessLimitOrder(setupOrder);
34     Assert.AreEqual(1, result.Count());
35     Assert.AreEqual(true, result.First().Placed);
36
37     setupOrder = new Order() { Price = 105, Quantity = 50,
        Side = OrderSide.Sell, Type = OrderType.LimitOrder,
        UserID = "Test" };
38     result = lob.ProcessLimitOrder(setupOrder);
39     Assert.AreEqual(1, result.Count());
40     Assert.AreEqual(true, result.First().Placed);
41
42     setupOrder = new Order() { Price = 94, Quantity = 30,
        Side = OrderSide.Buy, Type = OrderType.LimitOrder,
        UserID = "Test" };
43     result = lob.ProcessLimitOrder(setupOrder);
44     Assert.AreEqual(1, result.Count());
45     Assert.AreEqual(true, result.First().Placed);
46
47     setupOrder = new Order() { Price = 96, Quantity = 50,
        Side = OrderSide.Buy, Type = OrderType.LimitOrder,
        UserID = "Test" };
48     result = lob.ProcessLimitOrder(setupOrder);
49     Assert.AreEqual(1, result.Count());
50     Assert.AreEqual(true, result.First().Placed);
51
52     setupOrder = new Order() { Price = 97, Quantity = 40,
        Side = OrderSide.Buy, Type = OrderType.LimitOrder,
        UserID = "Test" };
53     result = lob.ProcessLimitOrder(setupOrder);
54     Assert.AreEqual(1, result.Count());
55     Assert.AreEqual(true, result.First().Placed);
56

```

```

57     setupOrder = new Order() { Price = 97, Quantity = 10,
        Side = OrderSide.Buy, Type = OrderType.LimitOrder,
        UserID = "Test" };
58     result = lob.ProcessLimitOrder(setupOrder);
59     Assert.AreEqual(1, result.Count());
60     Assert.AreEqual(true, result.First().Placed);
61
62     setupOrder = new Order() { Price = 97, Quantity = 10,
        Side = OrderSide.Buy, Type = OrderType.LimitOrder,
        UserID = "Test" };
63     result = lob.ProcessLimitOrder(setupOrder);
64     Assert.AreEqual(1, result.Count());
65     Assert.AreEqual(true, result.First().Placed);
66
67
68     //ACT
69     var marketOrder = new Order()
70     {
71
72         Quantity = 15,
73         Side = OrderSide.Buy,
74         Type =
75             OrderType.MarketOrder,
76         UserID = "Test2"
77     };
78
79     var testResult = lob.ProcessMarketOrder(marketOrder);
80
81     Assert.AreEqual(1, result.Count());
82
83     //ASSERT
84     Assert.AreEqual(2, lob.Asks[101].Count);
85     Assert.AreEqual(5, lob.Asks[101].ElementAt(0).Quantity);
86     Assert.AreEqual(10,
87         lob.Asks[101].ElementAt(1).Quantity);
88
89     Assert.AreEqual(1, lob.Asks[103].Count);
90     Assert.AreEqual(30,
91         lob.Asks[103].ElementAt(0).Quantity);
92
93     Assert.AreEqual(2, lob.Asks[104].Count);
94     Assert.AreEqual(15,
95         lob.Asks[104].ElementAt(0).Quantity);
96     Assert.AreEqual(5, lob.Asks[104].ElementAt(1).Quantity);
97
98     Assert.AreEqual(1, lob.Asks[105].Count);
99     Assert.AreEqual(50,
100         lob.Asks[105].ElementAt(0).Quantity);
101
102     Assert.AreEqual(3, lob.Bids[97].Count);

```

```

98     Assert.AreEqual(40, lob.Bids[97].ElementAt(0).Quantity);
99     Assert.AreEqual(10, lob.Bids[97].ElementAt(1).Quantity);
100    Assert.AreEqual(10, lob.Bids[97].ElementAt(2).Quantity);
101
102    Assert.AreEqual(1, lob.Bids[96].Count);
103    Assert.AreEqual(50, lob.Bids[96].ElementAt(0).Quantity);
104
105    Assert.AreEqual(1, lob.Bids[94].Count);
106    Assert.AreEqual(30, lob.Bids[94].ElementAt(0).Quantity);
107
108 }

```

B.2 Random Liquidity Taker

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using MarketSimulator.Contracts;
6  using MarketSimulator.Utills;
7
8  namespace MarketSimulator.Agents
9  {
10     public class RandomLiquidityTaker : IAgent
11     {
12         IRandomNumberGenerator _rng;
13         double _doNothingProbability;
14         double _maxOrderSize;
15
16         public RandomLiquidityTaker(IRandomNumberGenerator
17             randomNumberGenerator, double maxOrderSize,
18             double doNothingProbability)
19         {
20             _rng = randomNumberGenerator;
21             _doNothingProbability = doNothingProbability;
22             _maxOrderSize = maxOrderSize;
23         }
24         public Order GetNextAction(LimitOrderBookSnapshot
25             limitOrderBook)
26         {
27             Order order;
28             var size = Math.Ceiling(_maxOrderSize *
29                 _rng.GetRandomDouble());
30
31             var randomNumber = _rng.GetRandomDouble();
32
33             if (randomNumber < (0.5 -
34                 (_doNothingProbability / 2)))
35             {

```

```

31         //buymarketdorder
32         order = new Order() {Quantity = size,Type =
           OrderType.MarketOrder, Side =
           OrderSide.Buy };
33     }
34     else if (randomNumber > (0.5 +
           (_doNothingProbability / 2)))
35     {
36         order = new Order() { Quantity = size, Type
           = OrderType.MarketOrder, Side =
           OrderSide.Sell };
37     }
38     else
39     {
40         //do nothing
41         order = null;
42     }
43
44     return order;
45 }
46 }
47 }

```

B.3 Random Liquidity Maker

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using MarketSimulator.Utills;
6  using MarketSimulator.Contracts;
7  using MathNet.Numerics.Distributions;
8  using MathNet.Numerics.Random;
9
10 namespace MarketSimulator.Agents
11 {
12     public class RandomLiquidityMaker : IAgent
13     {
14         private IRandomNumberGenerator _rng;
15         private double _maxOrderSize;
16         private double _maxPriceDifferential;
17         private double _doNothingProbability;
18         private Normal _normal;
19         private int _decimals;
20
21         public RandomLiquidityMaker(IRandomNumberGenerator
           randomNumberGenerator, double maxOrderSize,
           double maxPriceDifferential, double
           doNothingProbability, Normal normalDist,int

```

```

22         decimals)
23     {
24         _rng = randomNumberGenerator;
25         _maxOrderSize = maxOrderSize;
26         _maxPriceDifferential = maxPriceDifferential;
27         _doNothingProbability = doNothingProbability;
28         _normal = normalDist;
29         _decimals = decimals;
30     }
31     public virtual Order
32     GetNextAction(LimitOrderBookSnapshot
33     limitOrderBook)
34     {
35         var size = Math.Ceiling(_maxOrderSize *
36         _rng.GetRandomDouble());
37         var priceDiff = _maxPriceDifferential *
38         Math.Abs(_normal.Sample());
39
40         Order order;
41
42         var randomNumber = _rng.GetRandomDouble();
43         if (randomNumber > (0.5 +
44         (_doNothingProbability/2)))
45         {
46             //buylimitorder
47             var price =
48             Math.Round(limitOrderBook.BestAskPrice.Value
49             - priceDiff, _decimals);
50             order = new Order { Price = price, Quantity
51             = size, Side = OrderSide.Buy, Type =
52             OrderType.LimitOrder, Valid = true};
53         }
54         else if (randomNumber < (0.5 -
55         (_doNothingProbability/2)))
56         {
57             //selllimitorder
58             var price =
59             Math.Round(limitOrderBook.BestBidPrice.Value
60             + priceDiff, _decimals);
61             order = new Order { Price = price, Quantity
62             = size, Side = OrderSide.Sell, Type =
63             OrderType.LimitOrder, Valid = true };
64         }
65         else
66         {
67             //donothing
68             order = null;
69         }
70     }

```

```

57         return order;
58     }
59 }
60 }

```

B.4 Heterogeneous trading rules agent

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using HetroTradingRules.TestParticipant.Console.Solvers;
7  using MarketSimulator.Contracts;
8  using MathNet.Numerics.Distributions;
9
10 namespace HetroTradingRules.TestParticipant.Console
11 {
12     public class HetroTradingRulesAgent
13     {
14         public int StockHeld { get; set; }
15         public double CashHeld { get; set; }
16
17         private decimal _fundamentalistWeighting;
18         private decimal _chartistWeighting;
19         private decimal _noiseWeighting;
20         private int _agentTimeHorizon;
21         private decimal _agentRiskAversionLevel;
22         private Random _randomGenerator;
23         private Normal _normal;
24         private Solver1D _solver;
25
26         public HetroTradingRulesAgent(double
27             fundamentalistWeighting, double
28             chartistWeighting, double noiseWeighting, double
29             referenceAgentTimeHorizon, double
30             referenceRiskAversionLevel, Random
31             randomGenerator, Solver1D solver)
32         {
33             _fundamentalistWeighting =
34                 (decimal)fundamentalistWeighting;
35             _chartistWeighting = (decimal)chartistWeighting;
36             _noiseWeighting = (decimal)noiseWeighting;
37             _agentTimeHorizon = (int)
38                 Math.Floor(referenceAgentTimeHorizon * ((1 +
39                     fundamentalistWeighting) / (1 +
40                     chartistWeighting)));
41             _agentRiskAversionLevel = (decimal)
42                 (referenceRiskAversionLevel * ((1 +

```

```

        fundamentalistWeighting) / (1 +
        chartistWeighting));
33     _randomGenerator = randomGenerator;
34     _solver = solver;
35 }
36
37 public Order GetAction(int timeStep, double[]
    spotPrice, double[] fundamentalValue, double[]
    noise, double? bestBid, double? bestAsk)
38 {
39     var fundamentalTimeHorizon = _agentTimeHorizon;
40     var lookBackTime = Math.Min(_agentTimeHorizon,
        timeStep)-1;
41
42     var normalisationTerm =
        _fundamentalistWeighting +
        _chartistWeighting + _noiseWeighting;
43
44     var fundamentalistTerm =
        (_fundamentalistWeighting *
        (decimal)Math.Log(fundamentalValue[timeStep
        - 1] / spotPrice[timeStep - 1])) /
        fundamentalTimeHorizon; //should the
        fundamental TH be added??
45
46
47     var averageReturn =
        CalculateAverageReturn(timeStep, spotPrice,
        lookBackTime);
48     var chartistTerm = _chartistWeighting *
        averageReturn;
49
50     var noiseTerm = _noiseWeighting * (decimal)
        noise[timeStep];
51
52
53     var expectedReturn = (fundamentalistTerm +
        chartistTerm + noiseTerm) /
        normalisationTerm;
54
55     var expectedPrice =
        Math.Round(spotPrice[timeStep-1] *
        Math.Exp((double)(expectedReturn *
        _agentTimeHorizon)),4);
56
57     var varianceOfPastReturns =
        CalculateVarianceOfPastReturns(timeStep,
        spotPrice, lookBackTime, averageReturn);
58

```

```

59     var comfortPriceAtCurrentHolding =
        Math.Round(FindRoot(p => GetStocksToHold(p,
            expectedPrice,
            (double)_agentRiskAversionLevel,
            (double)varianceOfPastReturns) - StockHeld,
            expectedPrice,
            0.0000001,0.01,expectedPrice), 4);
60
61     var maxPrice = expectedPrice;
62     var minPrice = FindRoot(p => p *
        (GetStocksToHold(p, expectedPrice,
            (double)_agentRiskAversionLevel,
            (double)varianceOfPastReturns) - StockHeld)
        - CashHeld, comfortPriceAtCurrentHolding,
        0.0000001,0.1,expectedPrice);
63
64     var drawnPrice =
        Math.Round(Math.Min(minPrice,maxPrice) +
            (Math.Abs(maxPrice-minPrice) *
            _randomGenerator.NextDouble()),4);
65
66     var order = new Order();
67
68     if (drawnPrice < comfortPriceAtCurrentHolding)
69     {
70         //buy
71         order.Side = OrderSide.Buy;
72
73         if (bestAsk.HasValue && drawnPrice >
            bestAsk.Value)
74         {
75             //buy market order
76             order.Quantity =
                GetStocksToHold(bestAsk.Value,
                    expectedPrice,
                    (double)_agentRiskAversionLevel,
                    (double)varianceOfPastReturns) -
                    StockHeld;
77             order.Type = OrderType.MarketOrder;
78             order.Price = bestAsk.Value;
79             StockHeld += (int)order.Quantity;
80             CashHeld -= order.Quantity *
                order.Price;
81         }
82         else
83         {
84             //buy limit order at drawnPrice
85             order.Quantity =
                GetStocksToHold(drawnPrice,
                    expectedPrice,

```

```

            (double)_agentRiskAversionLevel,
            (double)varianceOfPastReturns) -
            StockHeld;
86         order.Type = OrderType.LimitOrder;
87         order.Price = drawnPrice;
88     }
89 }
90 else if (drawnPrice >
          comfortPriceAtCurrentHolding)
91 {
92     //sell
93     order.Side = OrderSide.Sell;
94
95     if (bestBid.HasValue && drawnPrice <
        bestBid.Value)
96     {
97         //sell market order
98         order.Quantity = StockHeld -
            GetStocksToHold(bestBid.Value,
                expectedPrice,
                (double)_agentRiskAversionLevel,
                (double)varianceOfPastReturns);
99         order.Type = OrderType.MarketOrder;
100        order.Price = bestBid.Value;
101        StockHeld -= (int)order.Quantity;
102        CashHeld += order.Quantity *
            order.Price;
103    }
104
105    else
106    {
107        //sell limit order at drawnProce
108        order.Quantity = StockHeld -
            GetStocksToHold(drawnPrice,
                expectedPrice,
                (double)_agentRiskAversionLevel,
                (double)varianceOfPastReturns);
109        order.Type = OrderType.LimitOrder;
110        order.Price = drawnPrice;
111    }
112
113 }
114 else
115 {
116     //do nothing
117     order = null;
118 }
119
120 return order;
121 }

```

```

122
123     private static decimal CalculateAverageReturn(int
124         timeStamp, double[] spotPrice, int lookBackTime)
125     {
126         var total = 0m;
127         var count = 0;
128
129         for (int i = 1; i <= lookBackTime; i++)
130         {
131             count++;
132
133             total += (decimal)
134                 Math.Log(spotPrice[timeStep - i] /
135                     spotPrice[timeStep - i - 1]);
136         }
137
138         return total / count;
139     }
140
141     private static decimal
142     CalculateVarianceOfPastReturns(int timeStamp,
143     double[] spotPrice, int lookBackTime, decimal
144     averageReturn)
145     {
146         var total = 0m;
147         var count = 0;
148
149         for (int i = 1; i <= lookBackTime; i++)
150         {
151             count++;
152             total += (decimal)
153                 Math.Pow((double)((decimal)Math.Log(spotPrice[timeStep
154                     - i] / spotPrice[timeStep - i - 1]) -
155                     averageReturn), 2);
156         }
157
158         return total / count;
159     }
160
161     private double GetPortfolioWealth(double spotPrice)
162     {
163         return (StockHeld * spotPrice) + CashHeld;
164     }
165
166     private int GetStocksToHold(double spotPrice,
167     double expectedPrice, double riskAversionLevel,
168     double varianceOfPastReturns)
169     {

```

```

160         return (int) Math.Floor((Math.Log(expectedPrice
161             / spotPrice) / (riskAversionLevel *
162                 varianceOfPastReturns * spotPrice)));
163     }
164     private double FindRoot(Func<double, double>
165         function, double initialGuess, double error,
166         double lowerBound, double upperBound)
167     {
168         return _solver.solve(function, error,
169             initialGuess, lowerBound, upperBound);
170     }
171 }

```

B.5 Intra-day trading patterns agent

B.5.1 Agent

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7 using MarketSimulator.Contracts;
8
9 namespace IntradayTradingPatterns.TestParticipant.Console
10 {
11     public abstract class Agent
12     {
13         public double ExpectedAssetLiquidationValue { get;
14             set; }
15         public double
16             ExpectedAssetLiquidationValueOrderRange { get;
17                 set; }
18         public Random _random;
19         private int _maxOrderQuantity;
20         private string _name;
21         private readonly double _initialPropensity;
22         private readonly double _recency;
23         private readonly double _experimentation;
24         private readonly double _temperature;
25         private readonly string _group;
26
27         public double CurrentProfit { get; set; }
28
29         public string Name
30         {

```

```

28         get { return _name; }
29     }
30
31     BitArray timingChromosome;
32
33     public BitArray TimingChromosome
34     {
35         get { return timingChromosome; }
36         set { timingChromosome = value; }
37     }
38
39     public string Group
40     {
41         get { return _group; }
42     }
43
44     private List<ChromosomeLearning> _learningLog;
45
46     public Agent(Random randomNumberGenerator, int
47         maxOrderQuantity, string
48         name, IReadOnlyCollection<BitArray>
49         allTimingChromosomes, double initialPropensity,
50         double recency, double experimentation,
51         double temperature, string group)
52     {
53         _random = randomNumberGenerator;
54         _maxOrderQuantity = maxOrderQuantity;
55         _name = name;
56         _initialPropensity = initialPropensity;
57         _recency = recency;
58         _experimentation = experimentation;
59         _temperature = temperature;
60         _group = @group;
61
62         CurrentProfit = 0;
63
64         _learningLog = new List<ChromosomeLearning>();
65
66         foreach (var chromosome in allTimingChromosomes)
67         {
68             _learningLog.Add(new ChromosomeLearning()
69             {
70                 Probability =
71                     1d/allTimingChromosomes.Count,
72                 Propensity = initialPropensity,
73                 TimingChromosome = chromosome
74             });
75         }

```

```

72         TimingChromosome = _learningLog.OrderBy(l =>
73             l.Probability*_random.NextDouble()).First().TimingChromosome;
74     }
75
76     public MarketSimulator.Contracts.Order
77     GetNextAction(MarketSimulator.Contracts.LimitOrderBookSnapshot
78     limitOrderBookSnapshot, int day, int
79     tradingPeriod)
80     {
81         if (!WillTradeInThisPeriod(day, tradingPeriod))
82         {
83             return null;
84         }
85
86         var randomDraw = (ExpectedAssetLiquidationValue
87             - ExpectedAssetLiquidationValueOrderRange) +
88             (_random.NextDouble() *
89             ExpectedAssetLiquidationValueOrderRange * 2);
90
91         var orderQuantity =
92             Math.Round(_random.NextDouble() *
93             _maxOrderQuantity,0);
94
95         Order order = new Order();
96         order.Price = randomDraw;
97         order.Quantity = orderQuantity;
98         order.UserID = _name;
99
100        if (randomDraw > ExpectedAssetLiquidationValue)
101        {
102            //sell
103            order.Side = OrderSide.Sell;
104            if (limitOrderBookSnapshot != null &&
105                limitOrderBookSnapshot.BestBidPrice !=
106                null && randomDraw <
107                limitOrderBookSnapshot.BestBidPrice)
108            {
109                //sell market order
110                order.Type = OrderType.MarketOrder;
111            }
112            else
113            {
114                //sell limit order
115                order.Type = OrderType.LimitOrder;
116            }
117        }
118        else
119        {
120            //buy

```

```

110         order.Side = OrderSide.Buy;
111         if (limitOrderBookSnapshot != null &&
            limitOrderBookSnapshot.BestAskPrice !=
            null && randomDraw >
            limitOrderBookSnapshot.BestAskPrice)
112         {
113             //buy market order
114             order.Type = OrderType.MarketOrder;
115         }
116         else
117         {
118             //buy limit order
119             order.Type = OrderType.LimitOrder;
120         }
121     }
122
123     order = FilterLimitOrders(order);
124
125     return order;
126 }
127
128 public virtual Order FilterLimitOrders(Order order)
129 {
130     return order;
131 }
132
133 public abstract bool WillTradeInThisPeriod(int day,
            int tradingPeriod);
134
135
136 public List<string> GetOrdersToCancel(int day, int
            tradingPeriod)
137 {
138     return new List<string>();
139 }
140
141 public void RandomizeTimingChromosome(Random
            randomNumberGenerator)
142 {
143     for (int i = 0; i < TimingChromosome.Length;
            i++)
144     {
145         TimingChromosome[i] =
            randomNumberGenerator.Next() % 2 == 0;
146     }
147 }
148
149
150 public virtual void
            EvolveTimingChromosome(LearningMode

```

```

151         learningMode, Dictionary<BitArray, double> agents,
           double crossoverProbability, double
152         mutationProbability)
153     {
154         switch (learningMode)
155         {
156             case LearningMode.Random:
157                 RandomizeTimingChromosome(_random);
158                 break;
159             case LearningMode.GA:
160                 if (_random.NextDouble() <
161                     crossoverProbability)
162                 {
163                     //selection
164                     var chromosomes =
165                         agents.Select(a => new {a.Key,
166                             Rank =
167                                 a.Value*_random.NextDouble()});
168
169                     var chosenChromosome =
170                         chromosomes.OrderByDescending(c
171                             => c.Rank).First();
172
173                     //crossover
174                     var crossover =
175                         Math.Floor(_random.NextDouble()*TimingChromosome.Count);
176
177                     for (int i = 0; i < crossover; i++)
178                     {
179                         TimingChromosome[i] =
180                             chosenChromosome.Key[i];
181                     }
182
183                     //mutation
184                     for (int i = 0; i <
185                         TimingChromosome.Count; i++)
186                     {
187                         TimingChromosome[i] =
188                             _random.NextDouble() <
189                             mutationProbability
190                             ? !TimingChromosome[i]
191                             : TimingChromosome[i];
192                     }
193                 }
194                 break;
195             case LearningMode.MRE:
196                 //update propensities

```

```

188         for (int i = 0; i < _learningLog.Count;
189             i++)
190         {
191             double reward;
192             if
193                 (_learningLog[i].TimingChromosome
194                 == TimingChromosome)
195             {
196                 reward = CurrentProfit*(1 -
197                     _experimentation);
198             }
199             else
200             {
201                 reward =
202                     _learningLog[i].Propensity *
203                     (_experimentation /
204                     (_learningLog.Count - 1));
205             }
206             _learningLog[i].Propensity = (1 -
207                 _recency)*_learningLog[i].Propensity
208                 + reward;
209         }
210         //update probabilities
211         for (int i = 0; i < _learningLog.Count;
212             i++)
213         {
214             _learningLog[i].Probability =
215                 Math.Exp(_learningLog[i].Propensity/_temperature)/_learningLog[i].
216                 =>
217                 Math.Exp(1.Propensity/_temperature));
218         }
219         //select chromosome
220         TimingChromosome =
221             _learningLog.OrderByDescending(l =>
222                 l.Probability *
223                 _random.NextDouble()).First().TimingChromosome;
224         break;
225     default:
226         throw new
227             ArgumentOutOfRangeException("Unknown
228                 learning mode");
229     }
230     CurrentProfit = 0;
231 }

```

```

220
221     }
222
223     public class ChromosomeLearning
224     {
225         public BitArray TimingChromosome { get; set; }
226         public double Propensity { get; set; }
227         public double Probability { get; set; }
228     }
229
230     public enum LearningMode
231     {
232         Random,
233         GA,
234         MRE
235     }
236 }

```

B.5.2 Informed Agent

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Text;
6  using System.Threading.Tasks;
7  using MarketSimulator.Contracts;
8
9  namespace IntradayTradingPatterns.TestParticipant.Console
10 {
11     class InformedAgent : Agent, IAgent
12     {
13         bool _compete;
14
15
16         public InformedAgent(Random randomNumberGenerator,
17             int maxOrderQuantity, string name, bool compete,
18             List<BitArray> allTimingChromosomes, double
19             initialPropensity, double recency,
20             double experimentation, double temperature, string
21             @group)
22             : base(randomNumberGenerator, maxOrderQuantity,
23                 name, allTimingChromosomes, initialPropensity, recency,
24                 experimentation, temperature, group)
25         {
26             _compete = compete;
27         }
28     }
29 }

```

```

26     public override MarketSimulator.Contracts.Order
        FilterLimitOrders(MarketSimulator.Contracts.Order
            order)
27     {
28         if (order != null && !_compete && order.Type ==
            OrderType.LimitOrder)
29         {
30             return null;
31         }
32         else
33         {
34             return order;
35         }
36     }
37
38     public override bool WillTradeInThisPeriod(int day,
        int tradingPeriod)
39     {
40         return TimingChromosome[tradingPeriod];
41     }
42 }
43 }

```

B.5.3 Uninformed Agent

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace IntradayTradingPatterns.TestParticipant.Console
9 {
10     class UninformedAgent : Agent, IAgent
11     {
12
13         public UninformedAgent(Random
            randomNumberGenerator, int maxOrderQuantity,
            string name, List<BitArray>
            allTimingChromosomes, double initialPropensity,
            double recency, double experimentation, double
            temperature, string group)
14             : base(
15                 randomNumberGenerator, maxOrderQuantity,
                    name, allTimingChromosomes,
                    initialPropensity, recency,
16                 experimentation, temperature, group)
17         {
18

```

```

19     }
20
21     public override bool WillTradeInThisPeriod(int day,
22         int tradingPeriod)
23     {
24         return tradingPeriod ==
25             getIntFromBitArray(TimingChromosome);
26     }
27
28     //http://stackoverflow.com/questions/5283180/
29     //how-i-can-convert-bitarray-to-single-int
30     private int getIntFromBitArray(BitArray bitArray)
31     {
32         if (bitArray.Length > 32)
33             throw new ArgumentException("Argument
34                 length shall be at most 32 bits.");
35
36         int[] array = new int[1];
37         bitArray.CopyTo(array, 0);
38         return array[0];
39     }
40 }

```

B.5.4 Simulation coordinator

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Data.Odbc;
5 using System.IO;
6 using System.Linq;
7 using System.Text;
8 using System.Text.RegularExpressions;
9 using System.Threading.Tasks;
10 using MarketSimulator.Contracts;
11 using Microsoft.AspNet.SignalR.Client;
12 using Microsoft.AspNet.SignalR.Client.Hubs;
13
14 namespace IntradayTradingPatterns.TestParticipant.Console
15 {
16     class Program
17     {
18         private static LimitOrderBookSnapshot
19             _limitOrderBookSnapshot;
20
21         static void Main(string[] args)
22         {

```

```

22         System.Threading.Thread.Sleep(1000);
23
24         int NumberOfInformedAgents = 40;
25         int NumberOfUninformedAgents = 160;
26         int NumberOfDays = 2250;
27         int NumberOfTradingPeriods = 8;
28         double MinExpectedAssetValue = 20;
29         double MaxExpectedAssetValue = 120;
30         double MinExpectedAssetValueRange = 5;
31         double MaxExpectedAssetValueRange = 15;
32         int NumberOfDaysInLearningPeriod = 15;
33         bool InformedAgentsCompete = false;
34         int NumberOfAgentsInGroup = 40;
35         double crossOverProbability = 0.7;
36         double mutationProbability = 0.001;
37         double initialPropensity = 8000;
38         double recency = 0.10;
39         double experimentation = 0.15;
40         double temperature = 30;
41
42         var learningMode = LearningMode.MRE;
43
44         var random = new Random();
45
46         var connection = new
47             HubConnection(@"http://localhost:8080/signalr");
48         var hub = connection.CreateHubProxy("market");
49         connection.Start().Wait();
50
51         System.Console.WriteLine("Connected");
52
53         hub.On<LimitOrderBookSnapshot>("Update",
54             UpdateLimitOrderBook);
55
56         var subscribeResult =
57             hub.Invoke("SubscribeToDataFeed",
58                 "TestDriver");
59         subscribeResult.Wait();
60
61         var agents = new List<Agent>();
62
63         var allUnInformedTimingChromosomes =
64             GenerateAllTimingChromosomes(3);
65         var allInformedTimingChromosomes =
66             GenerateAllTimingChromosomes(8);
67
68         //init informed agents
69         for (int i = 0; i < NumberOfInformedAgents; i++)
70         {

```

```

66         var agent = new InformedAgent(random, 100,
           "InformedAgent" + i,
           InformedAgentsCompete,
           allInformedTimingChromosomes,
           initialPropensity, recency,
           experimentation, temperature,
           "informed");
67     agents.Add(agent);
68 }
69
70 //init uninformed agents
71 for (int i = 0; i < NumberOfUninformedAgents;
72     i++)
73 {
74     var agent = new UninformedAgent(random,
75         100, "UninformedAgent" + i,
76         allUnInformedTimingChromosomes,
77         initialPropensity, recency,
78         experimentation, temperature,
79         "uninformed" + i % 4);
80     agents.Add(agent);
81 }
82
83 var orderUpdater = new OrderUpdater(agents);
84 hub.On<OrderUpdate>("UpdateOrder",
85     orderUpdater.UpdateOrder);
86
87 var volume = new
88     double[NumberOfDays,NumberOfTradingPeriods];
89
90 //loop through days
91 for (int i = 0; i < NumberOfDays; i++)
92 {
93     System.Console.WriteLine(i);
94     var assetValue = MinExpectedAssetValue +
95         random.NextDouble() *
96         (MaxExpectedAssetValue - MinExpectedAssetValue);
97     orderUpdater.AssetValue = assetValue;
98
99     foreach (var agent in agents)
100    {
101        if (agent is InformedAgent)
102        {
103            agent.ExpectedAssetLiquidationValue
104                = assetValue;
105        }
106        else
107        {
108            agent.ExpectedAssetLiquidationValue
109                = MinExpectedAssetValue +

```

```

    random.NextDouble() *
    (MaxExpectedAssetValue -
    MinExpectedAssetValue);
98     }
99
100     agent.ExpectedAssetLiquidationValueOrderRange
        = MinExpectedAssetValueRange +
        random.NextDouble() *
        (MaxExpectedAssetValueRange -
        MinExpectedAssetValueRange);
101     }
102
103     //loop through trading periods
104     for (int j = 0; j < NumberOfTradingPeriods;
        j++)
105     {
106         //get next action from each agent
107         foreach (var agent in agents)
108         {
109             var ordersToCancel =
                agent.GetOrdersToCancel(i, j);
110
111             var order =
                agent.GetNextAction(_limitOrderBookSnapshot,
                i, j);
112
113             if (order != null)
114             {
115                 var r =
                    hub.Invoke<bool>("ProcessOrderInstruction",
                    order, agent.Name);
116                 r.Wait();
117                 //set agents profit/loss
118                 if (order.Type ==
                    OrderType.MarketOrder)
119                 {
120                     if (order.Side ==
                        OrderSide.Buy)
121                     {
122                         agent.CurrentProfit +=
                            (assetValue -
                            order.Price) *
                            order.Quantity;
123                     }
124                     else if (order.Side ==
                        OrderSide.Sell)
125                     {
126                         agent.CurrentProfit +=
                            (order.Price -
                            assetValue) *

```

```

127             order.Quantity;
128         }
129         volume[i, j] +=
130             order.Quantity;
131     }
132 }
133
134 //shuffle agents
135 agents.Shuffle();
136 }
137
138 if ((i + 1) % NumberOfDaysInLearningPeriod
139     == 0)
140 {
141     var agentByGroup = agents.GroupBy(a =>
142         a.Group);
143
144     var groups = new
145         Dictionary<string, Dictionary<BitArray,
146             double>>();
147
148     foreach (var group in agentByGroup)
149     {
150         if (!groups.ContainsKey(group.Key))
151         {
152             groups.Add(group.Key, new
153                 Dictionary<BitArray,
154                     double>());
155         }
156
157         foreach (var agent in group)
158         {
159             if (!groups[group.Key].
160                 ContainsKey(agent.TimingChromosome))
161             {
162                 groups[group.Key].
163                     Add(agent.TimingChromosome, agent.CurrentProfit);
164             }
165             else
166             {
167                 groups[group.Key][agent.TimingChromosome]
168                     += agent.CurrentProfit;
169             }
170         }
171     }
172 }
173
174 foreach (var agent in agents)
175 {

```

```

168         if (agent is UninformedAgent)
169         {
170             agent.EvolveTimingChromosome(learningMode,
171                 groups[agent.Group],
172                 crossOverProbability,
173                 mutationProbability);
174         }
175         else if (agent is InformedAgent)
176         {
177             agent.EvolveTimingChromosome(learningMode,
178                 groups[agent.Group],
179                 crossOverProbability,
180                 mutationProbability);
181         }
182     }
183     //clear trading book every day
184     var result = hub.Invoke("ClearAllTrades");
185     result.Wait();
186 }
187
188 private static string GetKey(BitArray
189     timingChromosome)
190 {
191     var key = "";
192     for (int i = 0; i < timingChromosome.Count; i++)
193     {
194         key += timingChromosome[i] ? "1" : "0";
195     }
196     return key;
197 }
198
199 private static List<BitArray>
200     GenerateAllTimingChromosomes(int length)
201 {
202     var allPossible = new List<BitArray>();
203     for (int i = 0; i < Math.Pow(2,length)-1; i++)
204     {
205         var intArray = new int[] {i};
206         var bitArray = new BitArray(intArray);
207         allPossible.Add(bitArray);
208     }
209 }
210

```

```

211
212     for (int i = 0; i < allPossible.Count; i++)
213     {
214         var newArray = new bool[length];
215         for (int j = 0; j < length; j++)
216         {
217             newArray[j] = allPossible[i][j];
218         }
219
220         allPossible[i] = new BitArray(newArray);
221     }
222
223     return allPossible;
224 }
225
226 private static void
227     UpdateLimitOrderBook(LimitOrderBookSnapshot data)
228 {
229     _limitOrderBookSnapshot = data;
230 }
231
232 public static class ExtensionMethods
233 {
234     public static void Shuffle<T>(this IList<T> list)
235     {
236         Random rng = new Random();
237         int n = list.Count;
238         while (n > 1)
239         {
240             n--;
241             int k = rng.Next(n + 1);
242             T value = list[k];
243             list[k] = list[n];
244             list[n] = value;
245         }
246     }
247 }
248
249
250 public class OrderUpdater
251 {
252     private readonly List<Agent> _agents;
253     public double AssetValue { get; set; }
254
255     public OrderUpdater(List<Agent> agents)
256     {
257         _agents = agents;
258     }
259

```

```

260     public void UpdateOrder(OrderUpdate order)
261     {
262         if (order.Matches != null &&
            order.Matches.Count > 0 && order.Order.Type
            == OrderType.LimitOrder)
263         {
264             var agentInvolved = _agents.Where(a =>
                a.Name == order.Order.UserID).First();
265
266             if (order.Order.Side == OrderSide.Buy)
267             {
268                 foreach (var match in order.Matches)
269                 {
270                     agentInvolved.CurrentProfit +=
                        (AssetValue - order.Order.Price)
                        * match.Quantity;
271                 }
272             }
273             else if (order.Order.Side == OrderSide.Sell)
274             {
275                 foreach (var match in order.Matches)
276                 {
277                     agentInvolved.CurrentProfit +=
                        (order.Order.Price -
                        AssetValue)* match.Quantity;
278                 }
279             }
280         }
281     }
282 }
283 }
284 }

```

B.6 Standard limit order book implementation

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using MarketSimulator.Contracts;
6
7  namespace MarketSimulator.LimitOrderBook
8  {
9      public class StandardLimitOrderBook : ILimitOrderBook
10     {
11         public StandardLimitOrderBook()
12         {
13             Bids = new SortedList<double, Queue<Order>>();
14             Asks = new SortedList<double, Queue<Order>>();

```

```

15     }
16
17     public SortedList<double, Queue<Order>> Bids
18     {
19         get;
20         private set;
21     }
22
23     public SortedList<double, Queue<Order>> Asks
24     {
25         get;
26         private set;
27     }
28
29     public Order BestBid
30     {
31         get { return Bids.Count != 0 ? new Order {
32             Price = Bids.Last().Key, Quantity =
33             Bids.Last().Value.Sum(b => b.Quantity) } :
34             null; }
35     }
36
37     public Order BestAsk
38     {
39         get { return Asks.Count != 0 ? new Order {
40             Price = Asks.First().Key, Quantity =
41             Asks.First().Value.Sum(a => a.Quantity)} :
42             null
43     }; }
44
45     public IEnumerable<OrderUpdate>
46     ProcessLimitOrder(Order order)
47     {
48         switch (order.Side)
49         {
50             case OrderSide.Buy:
51                 return ProcessBuyLimitOrder(order);
52             case OrderSide.Sell:
53                 return ProcessSellLimitOrder(order);
54             default:
55                 return new[] {new OrderUpdate()
56                     {
57                         Placed = false,
58                         Order = order,
59                         Message = "Order side not known"
60                     }
61                 };
62         }
63     }
64 }

```

```

58     private IEnumerable<OrderUpdate>
        ProcessSellLimitOrder(Order order)
59     {
60         if (Bids.Any() && order.Price < Bids.Last().Key)
61         {
62             //cross order
63             return new [] {new OrderUpdate() { Message
                = "Cross order", Order = order }};
64         }
65         if (!Asks.Keys.Contains(order.Price))
66         {
67             Asks.Add(order.Price, new Queue<Order>());
68         }
69
70         Asks[order.Price].Enqueue(order);
71
72         return new [] {new OrderUpdate()
73         {
74             Placed = true,
75             Order = order
76         }};
77     }
78
79     private IEnumerable<OrderUpdate>
        ProcessBuyLimitOrder(Order order)
80     {
81         if (Asks.Any() && order.Price >
            Asks.First().Key)
82         {
83             //cross order
84             return new [] { new OrderUpdate() { Message
                = "Cross order", Order = order } };
85         }
86         if (!Bids.Keys.Contains(order.Price))
87         {
88             Bids.Add(order.Price, new Queue<Order>());
89         }
90
91         Bids[order.Price].Enqueue(order);
92
93         return new [] {new OrderUpdate()
94         {
95             Placed = true,
96             Order = order
97         }};
98     }
99
100    public IEnumerable<OrderUpdate>
        AmendLimitOrder(Order order)
101    {

```

```

102         return AmendLimitOrder(order, false);
103     }
104
105     public IEnumerable<OrderUpdate>
106     AmendLimitOrder(Order order, bool cancel)
107     {
108         var amended = false;
109         var remove = false;
110
111         foreach (var bidGroup in Bids.Values)
112         {
113             foreach (var bid in bidGroup)
114             {
115                 if (bid.ID == order.ID)
116                 {
117                     if (cancel)
118                     {
119                         bid.Valid = false;
120
121                         if (bidGroup.Count() <= 1)
122                         {
123                             remove = true;
124                             Bids.Remove(bid.Price);
125                         }
126
127                         return new[] {new OrderUpdate()
128                                     {
129                                         Amended = false, Message =
130                                             "Order Canceled", Order =
131                                             bid, Placed=false}
132                                     };
133                     }
134
135                     if (bid.Price != order.Price)
136                     {
137                         bid.Valid = false;
138
139                         if (bidGroup.Count() <= 1)
140                         {
141                             remove = true;
142                             Bids.Remove(bid.Price);
143                         }
144
145                         return ProcessLimitOrder(order);
146                     }
147                     else
148                     {
149                         bid.Quantity = order.Quantity;
150                     }
151                 }
152             }
153         }
154
155         return new[] {new OrderUpdate() {

```

```

148             Amended = true, Message = "Order
149                 amended", Order =
150                 bid, Placed=true}
151         };
152     }
153 }
154 foreach (var askGroup in Asks.Values)
155 {
156     foreach (var ask in askGroup)
157     {
158         if (ask.ID == order.ID)
159         {
160             if (cancel)
161             {
162                 ask.Valid = false;
163
164                 if (askGroup.Count() <= 1)
165                 {
166                     remove = true;
167                     Asks.Remove(ask.Price);
168                 }
169
170                 return new[] {new OrderUpdate()
171                     {
172                         Amended = false, Message =
173                             "Order Canceled", Order =
174                             ask, Placed=false}
175                     };
176             }
177             if (ask.Price != order.Price)
178             {
179                 ask.Valid = false;
180                 if (askGroup.Count() <= 1)
181                 {
182                     remove = true;
183                     Asks.Remove(ask.Price);
184                 }
185                 return ProcessLimitOrder(order);
186             }
187             else
188             {
189                 ask.Quantity = order.Quantity;
190             }
191             return new[] {new OrderUpdate() {
192                 Amended = true, Message = "Order
193                     amended", Order =
194                     ask, Placed=true}
195             }
196         }
197     }
198 }

```

```

191         };
192     }
193 }
194 }
195
196     return new[] {new OrderUpdate() {
197         Message = "Order not found",
198         Order = order
199     }};
200 }
201
202 public IEnumerable<OrderUpdate>
203     ProcessMarketOrder(Order order)
204 {
205     switch (order.Side)
206     {
207         case OrderSide.Buy:
208             return ProcessBuyMarketOrder(order);
209         case OrderSide.Sell:
210             return ProcessSellMarketOrder(order);
211         default:
212             return new[] {new OrderUpdate()
213             {
214                 Placed = false,
215                 Order = order,
216                 Message = "Order side not known"
217             }};
218     }
219 }
220
221 private IEnumerable<OrderUpdate>
222     ProcessSellMarketOrder(Order order)
223 {
224     if (order.ExecutionValidity ==
225         OrderExecutionValidity.ForceOrKill)
226     {
227         if (order.Quantity > Bids.Sum(a =>
228             a.Value.Sum(l => l.Quantity)))
229         {
230             return new[] {new OrderUpdate()
231             {
232                 Placed = false,
233                 Order = order,
234                 Message = "FOK validity and not
235                     enough depth"
236             }};
237         }
238     }
239 }
240
241 var quantity = order.Quantity;

```

```

236
237     var matches = new List<Match>();
238     var orderUpdates = new List<OrderUpdate>();
239
240     while (quantity > 0 && Bids.Count != 0)
241     {
242         var prices = Bids.Keys;
243         for (int i = prices.Count - 1; i >= 0; i--)
244         {
245             var price = prices[i];
246             var orders = Bids[price];
247
248             while (orders.Any() && quantity > 0)
249             {
250                 var nextOrder = orders.Peek();
251                 if (!nextOrder.Valid)
252                 {
253                     orders.Dequeue();
254                     continue;
255                 }
256
257                 if (nextOrder.Quantity > quantity)
258                 {
259                     nextOrder.Quantity -= quantity;
260                     matches.Add(new Match() { Price
261                         = price, Quantity = quantity
262                         });
263                     orderUpdates.Add(new
264                         OrderUpdate()
265                         {
266                             Order = nextOrder,
267                             Matches = new List<Match> {
268                                 new Match() { Price =
269                                     price, Quantity =
270                                     quantity } }
271                         });
272                     quantity = 0;
273                 }
274                 if (nextOrder.Quantity <= quantity)
275                 {
276                     quantity -= nextOrder.Quantity;
277                     matches.Add(new Match() { Price
278                         = price, Quantity =
279                         nextOrder.Quantity });
280                     orderUpdates.Add(new
281                         OrderUpdate()
282                         {
283                             Order = nextOrder,
284                             Matches = new List<Match> {
285                                 new Match() { Price =

```

```

276         price, Quantity =
277             nextOrder.Quantity } }
278     });
279     orders.Dequeue();
280 }
281     if (!orders.Any()) Bids.Remove(price);
282 }
283 }
284
285     orderUpdates.Add(new OrderUpdate()
286     {
287         Placed = true,
288         Order = order,
289         Matches = matches
290     });
291
292     return orderUpdates;
293 }
294
295 private IEnumerable<OrderUpdate>
296 ProcessBuyMarketOrder(Order order)
297 {
298     if (order.ExecutionValidity ==
299         OrderExecutionValidity.ForceOrKill)
300     {
301         if (order.Quantity > Asks.Sum(a =>
302             a.Value.Sum(l => l.Quantity)))
303         {
304             return new[] {new OrderUpdate()
305             {
306                 Placed = false,
307                 Order = order,
308                 Message = "FOK validity and not
309                     enough depth"
310             }};
311         }
312     }
313
314     var quantity = order.Quantity;
315
316     var matches = new List<Match>();
317     var orderUpdates = new List<OrderUpdate>();
318
319     while (quantity > 0 && Asks.Count != 0)
320     {
321         var prices = Asks.Keys;
322         for (int i = 0; i < prices.Count; i++)

```

```

320     {
321         var price = prices[i];
322         var orders = Asks[price];
323
324         while (orders.Any() && quantity > 0)
325         {
326             var nextOrder = orders.Peek();
327             if (!nextOrder.Valid)
328             {
329                 orders.Dequeue();
330                 continue;
331             }
332
333             if (nextOrder.Quantity > quantity)
334             {
335                 nextOrder.Quantity -= quantity;
336                 matches.Add(new Match() { Price
337                     = price, Quantity = quantity
338                 });
339                 orderUpdates.Add(new
340                     OrderUpdate()
341                 {
342                     Order = nextOrder,
343                     Matches = new List<Match> {
344                         new Match() { Price =
345                             price, Quantity =
346                             quantity} }
347                 });
348                 quantity = 0;
349             }
350             if (nextOrder.Quantity <= quantity)
351             {
352                 quantity -= nextOrder.Quantity;
353                 matches.Add(new Match() { Price
354                     = price, Quantity =
355                     nextOrder.Quantity });
356                 orderUpdates.Add(new
357                     OrderUpdate()
358                 {
359                     Order = nextOrder,
360                     Matches = new List<Match> {
361                         new Match() { Price =
362                             price, Quantity =
363                             nextOrder.Quantity } }
364                 });
365                 orders.Dequeue();
366             }
367
368             if (!orders.Any())
369                 Asks.Remove(price);

```

```

357         }
358     }
359 }
360
361     orderUpdates.Add(new OrderUpdate()
362     {
363         Placed = true,
364         Order = order,
365         Matches = matches
366     });
367
368     return orderUpdates;
369 }
370
371 public IEnumerable<OrderUpdate> CancelOrder(Order
372     order)
373 {
374     return AmendLimitOrder(order, true);
375 }
376 }
377 }

```

B.7 SignalR Communications Module implementation

```

1 using System;
2 using System.Collections.Concurrent;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7 using MarketSimulator.Contracts;
8 using Microsoft.AspNet.SignalR;
9 using Microsoft.AspNet.SignalR.Hubs;
10 using Microsoft.Owin.Hosting;
11 using Owin;
12
13 namespace MarketSimulator.CommunicationsModule
14 {
15     public class SignalRCommunicationsHandler : IDataFeed,
16         ITradeInterface
17     {
18         private readonly static
19             Lazy<SignalRCommunicationsHandler> _instance =
20             new Lazy<SignalRCommunicationsHandler>(() => new
21                 SignalRCommunicationsHandler());
22         private readonly Lazy<IHubConnectionContext>
23             _clientsInstance = new

```

```

19         Lazy<IHubConnectionContext>(() =>
20             GlobalHost.ConnectionManager.GetHubContext<Market>().Clients);
21
22     public static SignalRCommunicationsHandler Instance
23         {get {return _instance.Value;}}
24
25     public ConcurrentDictionary<string, string>
26         ConnectionIDs = new
27         ConcurrentDictionary<string, string>();
28     public List<string> DataListeners = new
29         List<string>();
30
31     public SignalRCommunicationsHandler()
32     {
33         WebApplication.Start<Startup>(@"http://localhost:8080");
34         Console.WriteLine("Server running on {0}",
35             "unknown");
36     }
37
38     public bool SubscribeToDataFeed(string userID)
39     {
40         //implemented in market hub below
41         throw new NotImplementedException();
42     }
43
44     public bool UnsubscribeFromDataFeed(string userID)
45     {
46         //implemented in market hub below
47         throw new NotImplementedException();
48     }
49
50     public bool PushUpdate(ILimitOrderBook
51         limitOrderBook)
52     {
53         var orderBookSnapshot = new
54             LimitOrderBookSnapshot();
55         orderBookSnapshot.BestAskPrice =
56             limitOrderBook.BestAsk != null ?
57             (double?)limitOrderBook.BestAsk.Price : null;
58         orderBookSnapshot.BestAskQuantity =
59             limitOrderBook.BestAsk != null ?
60             (double?)limitOrderBook.BestAsk.Quantity :
61             null;
62         orderBookSnapshot.BestBidPrice =
63             limitOrderBook.BestBid != null ?
64             (double?)limitOrderBook.BestBid.Price : null;
65         orderBookSnapshot.BestBidQuantity =
66             limitOrderBook.BestBid != null ?
67             (double?)limitOrderBook.BestBid.Quantity :

```

```

51         null;
52     var dataConnections = ConnectionIDs.Where(c =>
53         DataListeners.Contains(c.Key)).Select(c =>
54             c.Value);
55     foreach (var connID in dataConnections)
56     {
57         _clientsInstance.Value.Client(connID).Update(orderBookSnapshot);
58     }
59     return true;
60 }
61
62 public event ProcessOrderHandler OnOrder;
63
64 public bool ProcessOrderInstruction(Order order,
65     string userID)
66 {
67     OnOrder(order, userID);
68     return true;
69 }
70
71 public bool PushOrderInstructionUpdate(OrderUpdate
72     order, string userID)
73 {
74     var connID = ConnectionIDs[userID];
75     _clientsInstance.Value.Client(connID).UpdateOrder(order);
76
77     return true;
78 }
79
80 class Startup
81 {
82     public void Configuration(IApplicationBuilder app)
83     {
84         app.MapHubs();
85     }
86 }
87
88 public class Market : Hub
89 {
90     SignalRCommunicationsHandler _commsHandler;
91
92     public Market() :
93         this(SignalRCommunicationsHandler.Instance) { }

```

```

94     public Market(SignalRCommunicationsHandler
          commsHandler)
95     {
96         _commsHandler = commsHandler;
97     }
98     public bool ProcessOrderInstruction(Order order,
          string userID)
99     {
100         RegisterUserID(userID, Context.ConnectionId);
101         return
          _commsHandler.ProcessOrderInstruction(order, userID);
102     }
103     public bool SubscribeToDataFeed(string userID)
104     {
105         RegisterUserID(userID, Context.ConnectionId);
106         if
          (!_commsHandler.DataListeners.Contains(userID))
107             _commsHandler.DataListeners.Add(userID);
108         return true;
109     }
110
111     public bool UnsubscribeFromDataFeed(string userID)
112     {
113         if
          (_commsHandler.DataListeners.Contains(userID))
114             _commsHandler.DataListeners.Remove(userID);
115         return true;
116     }
117
118     private void RegisterUserID(string userID, string
          connectionID)
119     {
120         _commsHandler.ConnectionIDs.AddOrUpdate(userID, connectionID,
          (oldValue, newValue) => newValue);
121     }
122
123     public void Ping()
124     {
125         Console.WriteLine("Ping called");
126         Clients.All.pong();
127     }
128 }
129 }
130 }

```

B.8 The Order data structure and enumerations

```

1         public class Order
2     {
3         public string ID { get; set; }

```

```

4     public string UserID { get; set; }
5     public OrderType Type { get; set; }
6     public OrderSide Side { get; set; }
7     public double Quantity { get; set; }
8     public double Price { get; set; }
9     public double? StopPrice { get; set; }
10    public OrderExecutionValidity ExecutionValidity {
        get; set; }
11    public OrderTimeValidity TimeValidity { get; set; }
12    public DateTime? ValidUntil { get; set; }
13    public Boolean Valid { get; set; }
14
15    public Order()
16    {
17        ID = Guid.NewGuid().ToString();
18        Valid = true;
19    }
20 }
21
22 public enum OrderType
23 {
24     LimitOrder,
25     MarketOrder,
26     StopLimitOrder,
27     StopMarketOrder,
28     Cancel,
29     Amend
30 }
31
32 public enum OrderSide
33 {
34     Buy,
35     Sell
36 }
37
38 public enum OrderExecutionValidity
39 {
40     ExecuteAndEliminate,
41     ForceOrKill
42 }
43
44 public enum OrderTimeValidity
45 {
46     GoodTillTime,
47     GoodTillCancelled
48 }

```

Bibliography

- J. Anderson and M. Evans. Intelligent agent modelling for natural resource management. *Mathematical and Computer Modelling*, 20(8):109 – 119, 1994. ISSN 0895-7177. doi: [http://dx.doi.org/10.1016/0895-7177\(94\)90235-6](http://dx.doi.org/10.1016/0895-7177(94)90235-6). URL <http://www.sciencedirect.com/science/article/pii/0895717794902356>.
- W Brian Arthur. *Asset pricing under endogenous expectations in an artificial stock market*. PhD thesis, Brunel University, London, 1996.
- Per Bak, Maya Paczuski, and Martin Shubik. Price variations in a stock market with many agents. *Physica A: Statistical Mechanics and its Applications*, 246(3):430–453, 1997.
- Kai Cao, Xiao Feng, and Hui Wan. Applying agent-based modeling to the evolution of eco-industrial systems. *Ecological Economics*, 68(11):2868–2876, 2009.
- Filippo Castiglione. Agent based modeling, 2006. URL http://www.scholarpedia.org/article/Agent_based_modeling.
- Nicholas T Chan, Blake LeBaron, Andrew W Lo, Tomaso Poggio, Andrew W Lo Yy, and Tomaso Poggio Zz. Agent-based models of financial markets: A comparison with experimental markets. 1999.
- Su Chen, Chen Hu, and Yijia Zhou. Order book simulator and optimal liquidation strategies. 2010.
- Yoonsik Cheon and Gary T Leavens. A simple and practical approach to unit testing: The jml and junit way. In *ECOOP 2002 Object-Oriented Programming*, pages 231–255. Springer, 2002.
- Carl Chiarella, Giulia Iori, and Josep Perelló. The impact of heterogeneous trading rules on the limit order book and order flows. *Journal of Economic Dynamics and Control*, 33(3):525–537, 2009.
- Chicago Mercantile Exchange. The Art of Hand Signals. pages 100–107.
- Rama Cont. Volatility clustering in financial markets: empirical facts and agent-based models. In *Long memory in economics*, pages 289–309. Springer, 2007.

- David Easley and Maureen O'hara. Price, trade size, and information in securities markets. *Journal of Financial economics*, 19(1):69–90, 1987.
- JD Farmer, Paolo Patelli, and II Zovko. The predictive power of zero intelligence in financial markets. 2004. URL <http://www.pnas.org/content/102/6/2254.short>.
- Thierry Foucault, Ohad Kadan, and Eugene Kandel. Limit order book as a market for liquidity. *Review of Financial Studies*, 18(4):1171–1217, 2005.
- Daniel Friedman. The double auction market institution: A survey. *The Double Auction Market: Institutions, Theories, and Evidence*, 14:3–25, 1993.
- Lei Gao, Bohdan Durnota, Yongsheng Ding, and Hua Dai. An agent-based simulation system for evaluating gridding urban management strategies. *Knowledge-Based Systems*, 26:174–184, 2012.
- Paul Glasserman. *Monte Carlo methods in financial engineering*, volume 53. Springer, 2004.
- Dhananjay K Gode and Shyam Sunder. Allocative efficiency of markets with zero-intelligence traders: Market as a partial substitute for individual rationality. *Journal of political economy*, pages 119–137, 1993.
- Martin D Gould, Mason A Porter, Stacy Williams, Mark McDonald, Daniel J Fenn, and Sam D Howison. Limit order books. *Quantitative Finance*, 13(11): 1709–1742, 2013.
- Ian Hickson. The Web Socket protocol, 2010. URL <http://tools.ietf.org/html/draft-hixie-thewebsocketprotocol-75>.
- Kiyoshi Izumi and Kazuhiro Ueda. Phase transition in a foreign exchange market-analysis based on an artificial market approach. *Evolutionary Computation, IEEE Transactions on*, 5(5):456–470, 2001.
- Johannesburg Stock Exchange. Equities Trading Manual, 2008.
- Johannesburg Stock Exchange. JSE Equity Market and Trading Functionality Overview, 2010.
- Brian D Kluger and Mark E McBride. Intraday trading patterns in an intelligent autonomous agent-based stock market. *Journal of Economic Behavior & Organization*, 79(3):226–245, 2011.
- Peter Kratz and Torsten Schöneborn. Optimal liquidation in dark pools. In *EFA 2009 Bergen Meetings Paper*, 2013.
- Eric Lavesson. *Writing Testable Software*. PhD thesis, Mid Sweden University, 2012.

- Charles M Macal and Michael J North. Tutorial on agent-based modelling and simulation. *Journal of Simulation*, 4(3):151–162, 2010.
- Adam MacKenzie, John O Miller, Raymond R Hill, and Stephen P Chambal. Application of agent based modelling to aircraft maintenance manning and sortie generation. *Simulation Modelling Practice and Theory*, 20(1):89–98, 2012.
- Robert E Marks. Breeding hybrid strategies: Optimal behaviour for oligopolists. *Journal of Evolutionary Economics*, 2(1):17–38, 1992.
- Robert C Martin. The dependency inversion principle. *C++ Report*, 8(6):61–66, 1996.
- Sergei Maslov. Simple model of a limit order-driven market. *Physica A: Statistical Mechanics and its Applications*, 278(3):571–578, 2000.
- MathDotNet. Math.NET Numerics. URL <http://numerics.mathdotnet.com/>.
- Adam J McLane, Christina Semeniuk, Gregory J McDermid, and Danielle J Marceau. The role of agent-based models in wildlife ecology and management. *Ecological Modelling*, 222(8):1544–1556, 2011.
- Microsoft. Microsoft .NET, 2013. URL <http://www.microsoft.com/net>.
- Microsoft. SortedList Class, 2014. URL [http://msdn.microsoft.com/en-us/library/system.collections.sortedlist\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/system.collections.sortedlist(v=vs.110).aspx).
- Barry L Nelson, John S Carson, and Jerry Banks. *Discrete event system simulation*. Prentice hall, 2001.
- James Nicolaisen, Valentin Petrov, and Leigh Tesfatsion. Market power and efficiency in a computational electricity market with discriminatory double-auction pricing. *Evolutionary Computation, IEEE Transactions on*, 5(5):504–523, 2001.
- Anna A Obizhaeva and Jiang Wang. Optimal trading strategy and supply/demand dynamics. *Journal of Financial Markets*, 16(1):1–32, 2013.
- OWIN Project. Open Web Interface for .NET. URL <http://owin.org/>.
- Keith Pilbeam. The profitability of trading in the foreign exchange market: chartists, fundamentalists, and simpletons. *Oxford Economic Papers*, pages 437–452, 1995.
- Craig Pirrong. Upstairs, downstairs: Electronic vs. open outcry exchanges. In *EFA 2003 Annual Conference Paper*, number 203, 2003.
- Marc Potters and Jean-Philippe Bouchaud. More statistical properties of order books and price impact. *Physica A: Statistical Mechanics and its Applications*, 324(1):133–140, 2003.

- John W Pratt. Risk aversion in the small and in the large. *Econometrica: Journal of the Econometric Society*, pages 122–136, 1964.
- Zong Qi. Simulation of Multi-Agents in Financial Market. Technical report, 2009.
- Marco Raberto, Silvano Cincotti, Sergio M Focardi, and Michele Marchesi. Agent-based simulation of a financial market. *Physica A: Statistical Mechanics and its Applications*, 299(1):319–327, 2001.
- Omid Roozmand, Nasser Ghasem-Aghae, Gert Jan Hofstede, Mohammad Ali Nematbakhsh, Ahmad Baraani, and Tim Verwaart. Agent-based modeling of consumer decision making process based on power distance and personality. *Knowledge-Based Systems*, 24(7):1075–1095, October 2011. ISSN 09507051. doi: 10.1016/j.knosys.2011.05.001. URL <http://linkinghub.elsevier.com/retrieve/pii/S0950705111000888>.
- Paul A Samuelson. Stochastic speculative price. *Proceedings of the National Academy of Sciences*, 68(2):335–337, 1971.
- Stephen R Schach. *An introduction to object-oriented systems analysis and design with UML and the unified process*. McGraw-Hill/Irwin, 2004.
- Thomas C Schelling. Dynamic models of segregation. *Journal of mathematical sociology*, 1(2):143–186, 1971.
- Schumpeter. Off-exchange share trading: Shining a light on dark pools, August 2011. URL <http://www.economist.com/blogs/schumpeter/2011/08/exchange-share-trading>.
- Andrei Shleifer and Lawrence H Summers. The noise trader approach to finance. *The Journal of Economic Perspectives*, pages 19–33, 1990.
- SignalR. ASP.NET SignalR, 2013. URL <http://signalr.net/>.
- D M Solis. C# and the .NET Framework. In *Illustrated C#*, pages 1–14. Apress, 2012.
- Ian Sommerville. Requirements Engineering Processes. In *Software Engineering*, pages 154–155. Pearson Education Limited, seventh ed edition, 2004a.
- Ian Sommerville. Object-oriented design. In *Software Engineering*, pages 330–331. Pearson Education Limited, seventh ed edition, 2004b.
- Jean-Christophe Soulié and Olivier Thébaud. Modeling fleet response in regulated fisheries: An agent-based approach. *Mathematical and computer modelling*, 44(5):553–564, 2006.
- Lawrence H Summers. Does the stock market rationally reflect fundamental values? *The Journal of Finance*, 41(3):591–601, 1986.

- Nicholas SP Tay and Scott C Linn. Fuzzy inductive reasoning, expectation formation and the behavior of security prices. *Journal of Economic Dynamics and Control*, 25(3):321–361, 2001.
- Xamarin. Mono, 2013. URL <http://www.mono-project.com/>.
- Jing Yang. *Agent-based Modelling and Market Microstructure*. PhD thesis, 2002.
- Bing Zhang, Yongliang Zhang, and Jun Bi. An adaptive agent-based modeling approach for analyzing the influence of transaction costs on emissions trading markets. *Environmental Modelling & Software*, 26(4):482–491, 2011a.
- Tao Zhang, Peer-Olaf Siebers, and Uwe Aickelin. Modelling electricity consumption in office buildings: An agent based approach. *Energy and Buildings*, 43(10):2882–2892, 2011b.