Using Simulated Limit Order Book Data To Evaluate Trade Execution Strategies

Andrew Wang

Advisor: Professor Mykhaylo Shkolnikov

Submitted in partial fulfillment

of the requirements for the degree of

Bachelor of Science in Engineering

Department of Operations Research and Financial Engineering

Princeton University

April 16, 2019

I hereby declare that I am the sole author of this thesis.

I authorize Princeton University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

_____

Andrew Wang

I further authorize Princeton University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

_____

Andrew Wang

Abstract

The electronic nature of markets today allows traders to infer information from limit order book (LOB) data and execute trades based on this information. Agents such as hedge funds are often interested in solving what is known as the optimal trade execution problem, where they attempt to minimize cost and market impact when buying or selling a large amount of inventory. In this thesis, historical LOB data is used to simulate a market environment in which trade execution strategies are tested. First, high-frequency LOB data from the Coinbase Pro cryptocurrency exchange is used to create a modified version of the queue-reactive model developed by Huang et al. (2013). A market simulator is built based on this model where market dynamics update in real-time in response to the agent's actions. The performances of common trade execution strategies are then evaluated in the simulated market environment. The findings from this thesis are relevant for institutions who are interested in using a simulated market environment to test trading strategies before executing them.

Acknowledgments

I would like to thank the many people who were instrumental to the writing of this thesis and to those who made my years at Princeton such a good time.

Thank you to Professor Mykhaylo Shkolnikov for all the help you provided throughout the year. Your expertise was essential for guiding me through the challenges I faced when writing this thesis.

Thank you to Jason and Steven for taking the time to edit my thesis. I hope you make the most of your senior year.

Thank you to Chris, Pat, Shamay, and Dora for making Dod 403 the best room on campus. Your presence made it an amazing place to live each day.

Thank you to the boys and girls of Clockwork for all the good times. Clockwork makes me so happy.

Thank you to Cap for welcoming me into your home. I'm fortunate to have been part of such an Illustrious community.

Thank you to TB for sticking with me through thick and thin. Even though we're going different places, I hope our paths cross again after graduation.

Finally, thank you to my family — Heather, Rebecca, Tiffany, Mom, and Dad — for supporting me my whole life. I wouldn't have made it here without you.

Lapidibus Aequus Durando

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 Problem Description

This thesis analyzes a problem in quantitative finance known as the optimal trade execution problem. Often, an agent such as a hedge fund seeks to buy or sell a large volume of a specific security in a short amount of time. The goal of the optimal trade execution problem is to form a strategy to execute this trade at the most favorable prices. In the case of acquiring inventory, the agent seeks to minimize cost, and in the case of selling inventory, the agent seeks to maximize revenue.

More formally, given a volume V of shares to buy and a time limit T, the agent seeks to limit the amount of money spent to acquire these shares. The actual prices at which the agent can acquire the shares depend on the supply of the shares at different prices. Thus, if V is high, the trade should not be executed in one order, since many of the shares are only available at higher prices. The agent can instead choose to split the order into smaller slices over time to realize lower prices overall. In general, there is a trade-off between how quickly the trade is executed and the market impact, so increased prices may be inevitable when T is low. However, various strategies have been adopted to minimize the price impact of large trades.

When evaluating the performance of trading strategies, the common industry benchmark of the Volume Weighted Average Price (VWAP) is used. If V is acquired throughout the time frame in chunks of size v_i and price p_i where $\sum_i v_i = V$, the VWAP is defined as

$$\frac{\sum_i v_i p_i}{\sum_i v_i}$$

1.2 The Limit Order Book

This problem can be illustrated more clearly by discussing the centralized limit order book (LOB). The LOB is the trading mechanism used by most exchanges around the world. The LOB for a security consists of the prices and volumes at which customers are willing to buy and sell the security (bids and asks respectively) where the possible prices occur at increments of the tick size. For example, part of the LOB for Coca-Cola (KO) on the NYSE where the tick size is \$0.01 may look like Table 1.1, in which the 5 best bids and asks are listed:

Table 1.1: Coca-Cola Limit Order Book

Bids		**Asks**	
Volume	Price	Price	Volume
1000	48.69	48.70	500
2000	48.68	48.71	1500
3000	48.67	48.72	3500
6000	48.63	48.75	2400
8000	48.58	48.80	10000

The price of the stock is considered to be \$48.695, which is the midpoint between the best bid and best ask prices.

Say, for example, that the hedge fund receives a signal to buy 7000 shares of Coca-Cola. It could do so by issuing a market buy order, in which the trade is executed

immediately at the best market price(s). If there is not enough supply at the best ask price, the order will progressively move up the order book until it is satisfied. In this case, it would buy 500 shares at \$48.70, 1500 shares at \$48.71, 3500 shares at \$48.72, and the remaining 1500 shares at \$48.75. In this case, the VWAP would be (500*48.70 + 1500*48.71 + 3500*48.72 + 1500*48.75)/7000 = \$48.723/share. It could also place a limit buy order, which is only executed at the specified price or better. For example, it could place a limit order for 7000 shares at \$48.67. This order would be appear in the bid side of the order book until it is matched with a sell order. The updated LOB is shown in Table 1.2.

Table 1.2: Coca-Cola Updated Limit Order Book

Bids		**Asks**	
Volume	Price	Price	Volume
1000	48.69	48.70	500
2000	48.68	48.71	1500
10000	48.67	48.72	3500
6000	48.63	48.75	2400
8000	48.58	48.80	10000

Exchanges typically institute a time priority policy, which means that orders submitted at the same price are executed in the order at which they arrived. Although the limit order has a maximum price at execution, it is not guaranteed to execute like the market order.

Of course, the agent could also place multiple market or limit orders over time to acquire the inventory while minimizing market impact. Splitting the trade into multiple orders forms the basis of many optimal trade execution strategies.

Chapter 2

Literature Review

2.1 Past Approaches to Analyzing Optimal Trade Execution

Various approaches have been taken in the past to tackle the optimal trading execution problem. Several studies solve the optimal trade execution problem under different mathematical models of the asset price movement. Almgren and Chriss (1999) formulate a theoretical solution for optimal trade execution that is based on trading off execution cost mean and variance. They develop a closed form solution that maximizes expected utility of the trade at a given level of risk aversion. Gatheral and Schied (2011) and Forsyth et al. (2012) expand upon this work by solving optimal trade execution under the assumption that the asset price follows a geometric Brownian motion.

Studies have also been conducted using historical LOB data. Several studies use machine learning techniques on high-frequency data from exchanges to develop optimal trade execution methods. Lim and Coggins (2005) use genetic algorithms to develop an optimal trading strategy using data from the Australian Stock Exchange. Spooner et al. (2018) apply reinforcement learning to optimize market making actions

using equities data from across various stock market exchanges. Nevmyvaka et al. (2006) use reinforcement learning to solve the problem for several stocks listed on the NASDAQ. These studies back-test the strategies on historical data, but assume that the agent's actions do not affect the future dynamics of the LOB.

This thesis seeks to evaluate trading strategies in a realistic market environment with the impact of the agent's actions taken into account. In order to do so, an LOB must be simulated that reflects realistic market behavior. Hollifield et al. (2004) and Bouchaud et al. (2002) perform statistical analysis on the empirical studies of LOBs across major exchanges. Although it is difficult to reflect all the properties of empirical LOBs, studies have modelled important aspects of LOB dynamics. Obizhaeva and Wang (2013) use a supply and demand framework of LOB behavior to formulate optimal trade execution strategies involving both discrete and continuous trades. Alfonsi and Schied (2007) expand upon this work, allowing for general LOB shapes specified by density functions.

2.2 LOB Queuing Model

Queuing models have shown promise in providing a simple model for LOB dynamics that reflect empirical properties. Smith et al. (2003) provide a model of the LOB where orders follow a Poisson arrival process, allowing for statistical predictions on market behaviors. Cont et al. (2010) model the LOB as a Markovian queuing system and estimate parameters based on data from the Tokyo stock exchange. They then use Laplace transform methods to estimate the conditional probabilities of market events based on the shape of the LOB, finding that the model provides insights on short-term LOB behavior. Other studies have used models similar to the queuing model where arrivals follow Poisson processes. For example, El Euch et al. (2018) model price jumps of an asset using a bi-dimensional Hawkes process. This model reflects several

desirable properties of modern electronic markets such as its endogenous nature, where orders are correlated and made in response to each other. Huang et al. (2013) develop a model where the LOB is a multi-dimensional queuing system centered around the mid-price of the asset. This model accurately reproduces market behavior and lends itself easily to estimating parameters from market data. The model is described below:

The LOB is modelled as a $2K$-dimensional vector Q, where K is number of ticks on each side of the Markovian queuing system. Let p_0 be the reference price of the stock, which is the mid-price between two adjacent ticks. $[Q_k : k = -1 \ldots -K]$ represents the number of orders on the bid side $k + 0.5$ ticks below p_0 and $[Q_k : k = 1 \ldots K]$ represents the number of orders on the ask side $k - 0.5$ ticks above p_0. See Table 2.1 for the positions corresponding to the prices in the Coca-Cola LOB from Table 1.1, where $Q_k = 0$ at positions like $k = 4$ where the LOB is empty. Then,

$$Q(t) = (Q_{-K}(t), \ldots, Q_{-1}(t), Q_1(t), \ldots, Q_K(t))$$

is a continuous-time Markov jump process with a jump size equal to one.

Table 2.1: Queuing Model Positions with $p_0 = 48.695$

Bids		**Asks**	
k	Price	Price	k
-1	48.69	48.70	1
-2	48.68	48.71	2
-3	48.67	48.72	3
-7	48.63	48.75	6
-12	48.58	48.80	11

An event at position k occurs any time the number of shares offered at position k changes. It increases when a trader places a limit order at the price of position k and decreases when it is either consumed by a market order from the other side or a

limit order is cancelled.

The simplest model assumes that Q_k is independent for each k. Each jump when Q_k increases by 1 and $Q = q$ follows a Poisson arrival process with rate $\lambda_k^+(q)$ and each jump when Q_k decreases by 1 and $Q = q$ follows a Poisson arrival process with rate $\lambda_k^-(q)$. In order to model each Q_k as a positive integer with changes of size 1, Q_k represents the number of Average Event Sizes present at the LOB position. The AES is the average size of the absolute change in the number of shares at the queue position whenever an event occurs, where the AES at each position (AES_k) could differ. In this model, $Q_k = n$ would mean that there are $n * AES_k$ shares offered at position k.

p_0 is chosen to be a mid-price in between the best bid and ask prices. Let p_{-1} be the best bid price and p_1 be the best ask price. If p_{-1} and p_1 are an odd number of ticks apart, then

$$p_0 = (p_{-1} + p_1)/2$$

If p_{-1} and p_1 are an even number of ticks apart, p_0 could be picked to be $(p_{-1} + p_1)/2 + 0.5$ or $(p_{-1} + p_1)/2 - 0.5$. The one that is closest to the previous reference price is chosen.

When an event occurs that changes the best bid or ask, p_0 is updated. It can be seen that under certain conditions, this LOB model follows an ergodic Markov process. Historical LOB data can then be used to estimate AES_k and $\lambda_k^{\pm}(q)$ for each k.

In simulating the LOB, this thesis examines the assumptions made in this model using market exchange data. Modifications of the model are made to more accurately reflect observations from the data set, as described in Section 4.1. A market simulator is then built based on this modified model where trade execution strategies are tested.

Chapter 3

Data Source

3.1 Coinbase Exchange

To simulate real world securities, high frequency LOB data of cryptocurrency securities is collected from Coinbase Pro. Coinbase Pro (formerly known as Global Digital Asset Exchange or GDAX) is the advanced trading platform of Coinbase, which is an exchange founded in 2012 that brokers trades of many cryptocurrencies as well as fiat-currency to cryptocurrency exchanges. Major cryptocurrencies that are traded on Coinbase Pro include Bitcoin, Bitcoin Cash, Ethereum, Ethereum Classic, and Litecoin, while fiat-currencies are typically the USD or Euro. As of December 2018, Coinbase Pro has a total daily trading volume of about $70 million (CoinMarketCap (2018)). Coinbase Pro was chosen as the data source since it has readily accessible real-time LOB data that is freely available to developers from its API. It also features relatively liquid securities with large trading volumes.

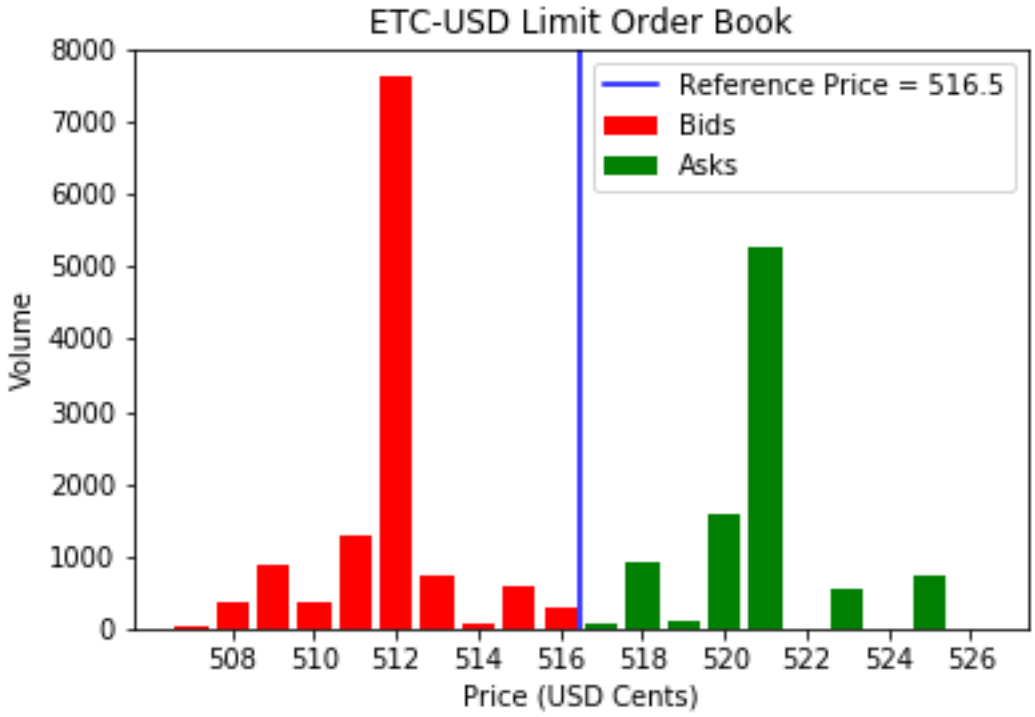
Although the LOB dynamics of the securities modelled in this paper may not entirely reflect those of other types of securities such as stocks, options, futures, etc., from other exchanges, the methodologies in this paper are adaptable to the LOB data of any relatively liquid security. The trading rules of Coinbase Pro are representative

of most exchanges around the world. Coinbase Pro allows limit and market orders with time priority matching. It also allows stop orders, which is an instruction to place a limit or market order when the price of the security reaches a specified price. In addition, Coinbase Pro differentiates between maker and taker orders. Taker orders are orders that fill immediately (such as market orders) and maker orders are orders that do not fill immediately and thus populate the LOB (such as limit orders). The fee structure, which is up to 0.3% for taker orders depending on the volume of the trade and 0% for maker orders encourages market making and therefore increases liquidity of the market. There are also rules to prevent self-trade, which is when the same trader acts as both the maker and taker for a trade. In addition, there are minimum and maximum orders for each security. Full trading rules for Coinbase Pro can be found on its website at `https://www.coinbase.com/legal/trading_rules?locale=en` (Coinbase (2018)).

3.2 Data Collection

Data is collected through the Coinbase Pro API using a Python library called CoPrA, which is an asynchronous web socket client (Python Software Foundation (2018)). Specifically, the "level2" channel is used, which first provides a snapshot of the LOB. It also provides updates at every position of the LOB whenever an order happens. See Listing A.2 for the code written for data collection. Using these updates, an internal LOB is maintained for use during data analysis. Data was collected for several days in the months of December and January for all the level2 updates on the ETC-USD LOB (Ethereum Classic for USD), which has a daily trading volume of around $2 million (CoinMarketCap (2018)). The initial LOB is shown in Figure 3.1. The 10 best bids and asks are shown, which is where the vast majority of updates are made, but updates are also recorded for prices further from p_0.

Figure 3.1: LOB Sample



This LOB is typical for ETC-USD, with most of the positions near p_0 filled. The bid and ask sides are relatively balanced, meaning that p_0 is stable. This pattern is not always the case, as the bid or ask side could have significantly more volume when the price shifts. A sample of updates is shown in Figure 3.2. Updates consist of the side, price, amount, and time. An update with an amount "0" means that the liquidity at the price is completely consumed. As can be seen, the updates are given to the nearest millisecond and the majority of the updates occur near p_0. The remainder of the thesis is based on the analysis of trading data from December 28-30, 2018 that includes almost all the updates of ETC-USD during that time period (there are small gaps of a few minutes due to starting and stopping data collection).

Figure 3.2: First 20 LOB Updates for December 30, 2018


```
{"side": "sell", "price": "5.17000000", "amount": "0", "time": "2018-12-30T05:04:16.909Z"}
{"side": "sell", "price": "5.18000000", "amount": "990.02649536", "time": "2018-12-30T05:04:16.922Z"}
{"side": "buy", "price": "5.13000000", "amount": "801.86038986", "time": "2018-12-30T05:04:16.937Z"}
{"side": "sell", "price": "5.18000000", "amount": "839.62649536", "time": "2018-12-30T05:04:21.932Z"}
{"side": "buy", "price": "5.15000000", "amount": "501.24106151", "time": "2018-12-30T05:04:21.938Z"}
{"side": "buy", "price": "5.16000000", "amount": "385.81602763", "time": "2018-12-30T05:04:21.998Z"}
{"side": "buy", "price": "5.12000000", "amount": "7676.41015625", "time": "2018-12-30T05:04:25.797Z"}
{"side": "buy", "price": "5.15000000", "amount": "1.24106151", "time": "2018-12-30T05:04:31.438Z"}
{"side": "buy", "price": "5.16000000", "amount": "311.6601938", "time": "2018-12-30T05:04:31.444Z"}
{"side": "buy", "price": "5.13000000", "amount": "918.71038986", "time": "2018-12-30T05:04:31.500Z"}
{"side": "buy", "price": "5.15000000", "amount": "75.39689534", "time": "2018-12-30T05:04:31.508Z"}
{"side": "buy", "price": "5.16000000", "amount": "811.6601938", "time": "2018-12-30T05:04:31.752Z"}
{"side": "buy", "price": "5.13000000", "amount": "801.86038986", "time": "2018-12-30T05:04:32.690Z"}
{"side": "sell", "price": "5.20000000", "amount": "1066.49797749", "time": "2018-12-30T05:04:33.445Z"}
{"side": "sell", "price": "5.21000000", "amount": "5769.34324935", "time": "2018-12-30T05:04:33.533Z"}
{"side": "sell", "price": "5.18000000", "amount": "687.43052699", "time": "2018-12-30T05:04:34.522Z"}
{"side": "sell", "price": "5.21000000", "amount": "5269.34324935", "time": "2018-12-30T05:04:34.979Z"}
{"side": "sell", "price": "5.20000000", "amount": "1566.49797749", "time": "2018-12-30T05:04:35.305Z"}
{"side": "sell", "price": "5.20000000", "amount": "1066.49797749", "time": "2018-12-30T05:04:43.275Z"}
{"side": "sell", "price": "5.21000000", "amount": "5769.34324935", "time": "2018-12-30T05:04:43.568Z"}
```

Chapter 4

Modelling the LOB

4.1 The Modified Queuing Model

Using the data collected, a model for simulating the LOB is created. It is based on the one developed by Huang et al. (2013) but with several important modifications. The LOB is again represented as a $2K$-dimensional vector

$$Q(t) = (Q_{-K}(t), \dots Q_{-1}(t), Q_1(t), \dots, Q_K(t))$$

centered around p_0. Events at each position k cause either a positive or negative change in $Q_k(t)$. The magnitudes of the events are distributed exponentially with means μ_k^+ and μ_k^- respectively. That is, if a positive event occurs at time t with size e at position k, set $q_k(t) \leftarrow q_k(t) + e$, and if a negative event occurs at time t with size e at position k, set $q_k(t) \leftarrow (q_k(t) - e)^+$ where $b^+ = 0$ if $b < 0$ and b otherwise (in order to keep the queue size non-negative). An exception occurs when a negative event occurs at the best bid or ask price. It is assumed to be a market order that will consume liquidity from successive prices until it is filled. If the size of the event is larger than the volume at the best price, the leftover part of the order is filled with the next best price and so on. The arrivals of events are modelled as a

multivariate Poisson process. With positive and negative arrivals at each of the $2K$ positions, there are $4K$ such processes. The marginal arrival process at each position is a Poisson process with positive and negative arrivals having mean rates λ_k^+ and λ_k^- respectively. Arrivals are simulated in chunks of length t. Let $N_k^+(t)$ be the number of positive arrivals and $N_k^-(t)$ be the number of negative arrivals at position k during a time period of length t. Then, the arrivals have a $4K \times 4K$ correlation matrix R where $R_{i+,j+} = corr(N_i^+(t), N_j^+(t))$, $R_{i+,j-} = corr(N_i^+(t), N_j^-(t))$, $R_{i-,j+} = corr(N_i^-(t), N_j^+(t))$, and $R_{i-,j-} = corr(N_i^-(t), N_j^-(t))$.

There are several differences between this model and the queue-reactive model developed by Huang et al. (2013). Modifications were made to better fit observations from the data set. First, the sizes of positive and negative events are not just the AES's, but rather exponential random variables with means equal to the AES's. This adjustment was made because the event sizes exhibit considerable variation. Section 4.3 shows that the exponential distribution reasonably fits the distributions of event sizes. Although it may not fit the data completely, it is the easiest part of the model to adjust.

Second, the rate of arrival of events at a given position does not vary as the size of the queue changes. In the data set, the queue sizes varied widely (sometimes several dozens multiplies of the AES's) and no substantial changes in rates were observed when the queue sizes changed. It is possible that the rates differ in a predictable way as the queue sizes change, but there were not enough data points to observe reliable differences, especially in intermediate sizes where there was no data at all. This change also has the added benefit of simplifying the model greatly.

Third, arrival processes are modelled as a multivariate Poisson process instead of independent Poisson processes. In Section 4.4, the claim that the arrival rates at each position reasonably follow individual Poisson processes is validated. However, Section 4.5 finds that these processes are significantly correlated. A multivariate

Poisson process allows these correlations to be incorporated in the simulation.

4.2 Building An Abbreviated Order Book

To facilitate analysis, steps were taken to clean the raw data. Often in the data set, several updates that have the same magnitude, sign, and inter-arrival time occur in quick succession. These orders were likely made by a trader who split up a larger trade. As it should represent one event, orders that occur in quick succession (defined as occurring less than 0.01 seconds after the last) are combined if they have the same price and sign.

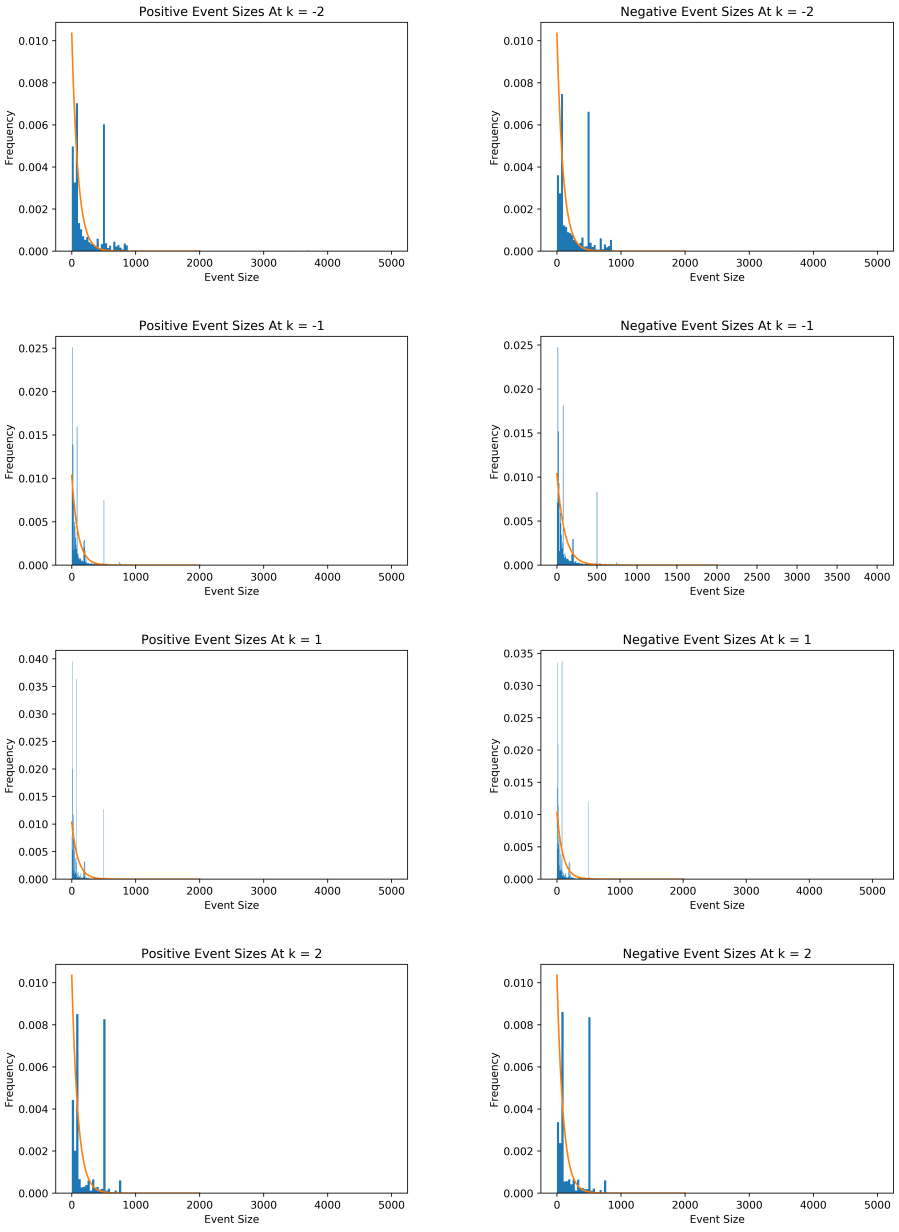
Although the full order book is maintained after each update, the queuing model only contains the first K prices on the bid and ask sides closest to p_0. Therefore, the parameters are only estimated for these positions. p_0 is maintained the same way as described in Section 2.2. If it changes, the data recording process is restarted. Analysis in the rest of the thesis is conducted for $K = 10$, which is wide enough to include the vast majority of events. See Listing A.3 for the code written to process the data and build the abbreviated order book.

4.3 Event Size Estimates

The sizes of the events at position k are distributed exponentially with means μ_k^+ and μ_k^-. A reasonable choice for μ_k^+ and μ_k^- is the AES for positive and negative events at each k. This section examines whether the event sizes can reasonably fitted with exponential distributions with rates equal to the AES's.

Histograms of event sizes at each position are shown in Figure 4.1. Exponential distributions with means equal to the AES's are overlaid on top. As can be seen, the exponential distributions fit the histograms relatively well. However, there are some notable discrepancies. One is the high frequency of orders of size 500. A possible

Figure 4.1: Event Sizes Compared to Exponential Distribution (4 Positions Closest to p_0)



explanation for this observation is that 500 may be a common size used by traders to split up large orders. There are also a non-negligible number of orders near 5000, since that is the upper limit order size imposed by Coinbase. Although it may be possible to more accurately model the dynamics of the ETC-USD LOB with these characteristics taken into account, they are not included in the model so that it can be more broadly applicable to other securities. The AES's are listed in Table 4.1 as $\mu_k^{\pm}$. See Listing A.4 for the code used to find the AES's. It can be seen that in general, as the price gets closer to p_0, $\mu_k^{\pm}$ decreases, but the number of events across the time period, $n_k^{\pm}$, increases. This higher rate of activity near p_0 makes sense, since market orders are executed at the best available prices and market makers are incentivized by the fee structure to provide liquidity around p_0.

Table 4.1: Event Size and Arrival Rate Parameter Estimates

	Positive Events				**Negative Events**			
k	n_k^+	μ_k^+	λ_k^+	$\mu_k^+ \cdot \lambda_k^+$	n_k^-	μ_k^-	λ_k^-	$\mu_k^- \cdot \lambda_k^-$
-10	406	601.41	0.0016	0.94	493	597.76	0.0019	1.14
-9	699	479.69	0.0027	1.29	739	446.6	0.0029	1.27
-8	1554	584.57	0.006	3.5	1602	552.68	0.0062	3.42
-7	5633	530.34	0.0217	11.53	5417	534.07	0.0209	11.16
-6	13115	520.95	0.0506	26.36	12852	531.86	0.0496	26.37
-5	21824	487.88	0.0842	41.08	21850	494.66	0.0843	41.7
-4	27620	427.19	0.1066	45.52	27578	433.09	0.1064	46.08
-3	33187	368.47	0.128	47.18	33214	367.8	0.1281	47.13
-2	47229	232.15	0.1822	42.3	45189	247.63	0.1743	43.17
-1	54254	104.11	0.2093	21.79	49177	104.8	0.1897	19.88
1	48246	93.79	0.1861	17.46	46787	89.82	0.1805	16.21
2	37702	226.64	0.1455	32.96	38310	229.59	0.1478	33.93
3	41409	305.04	0.1598	48.73	41912	302.37	0.1617	48.89
4	44948	328.57	0.1734	56.98	44616	333.46	0.1721	57.4
5	37880	318.37	0.1461	46.53	37105	325.12	0.1431	46.54
6	16320	360.54	0.063	22.7	15669	375.32	0.0604	22.69
7	3818	577.4	0.0147	8.5	3672	587.94	0.0142	8.33
8	1271	599.09	0.0049	2.94	1280	516.85	0.0049	2.55
9	602	428.87	0.0023	1	543	484.68	0.0021	1.02
10	548	366.15	0.0021	0.77	529	370.48	0.002	0.76

4.4 Arrival Rate Estimates

The event arrivals are modelled as a multivariate Poisson process, where the marginal processes at each position k have average arrival rates $\lambda_k^{\pm}$. First, the claim that individual event arrivals at each position follow Poisson processes is examined. To do so, inter-arrival times of the events are tested to see if they are exponentially distributed. Figures 4.2 and 4.3 show QQ-plots and histograms of the inter-arrival times compared to exponential distributions. From the QQ-plots, it can be seen that the inter-arrival times have heavy right tails compared to the exponential distribution. From the histograms, it can be seen that the exponential distribution fits the data well for the majority of the data points except for a small number of points on the right that comprise the heavy tails. Because the exponential distribution reasonably fits the inter-arrival times as a whole, the arrivals are modelled as a multivariate Poisson process for its desirable properties in simulation. We estimate $\lambda_k^{\pm}$ by taking the number of arrivals of the specified event ($N_k^{\pm}$) divided by the total time period. The code used to find the rates is found in Listing A.4. The average rates are reported in Table 4.1.

The products $\lambda_k^+ \cdot \mu_k^+$ and $\lambda_k^- \cdot \mu_k^-$ can be thought of as the average rate of impact of positive and negative events at position k. In other words, they are the average positive and negative rates at which the size of the queue drifts over time. One concern with this model is that the impact of the positive events may overwhelm the impact of the negative events over time. That is, if $\lambda_k^+ \cdot \mu_k^+ > \lambda_k^- \cdot \mu_k^-$, then $Q_k(t)$ is unbounded as $t \to \infty$, which is of course unrealistic. This situation could arise due to trade imbalances in the time frame of the data that we use to estimate the parameters. We compare $\lambda_k^+ \cdot \mu_k^+$ and $\lambda_k^- \cdot \mu_k^-$ in Table 4.1. As can be seen, some rates are indeed larger for positive events compared to negative events. However, because the positive and negative rates are nearly the same at each position, $Q_k(t)$ should not increase an unreasonable amount in a short simulation time frame.

Figure 4.2: Inter-Arrival Times for Positive Events Compared to Exponential Distribution (4 Positions Closest to p_0)

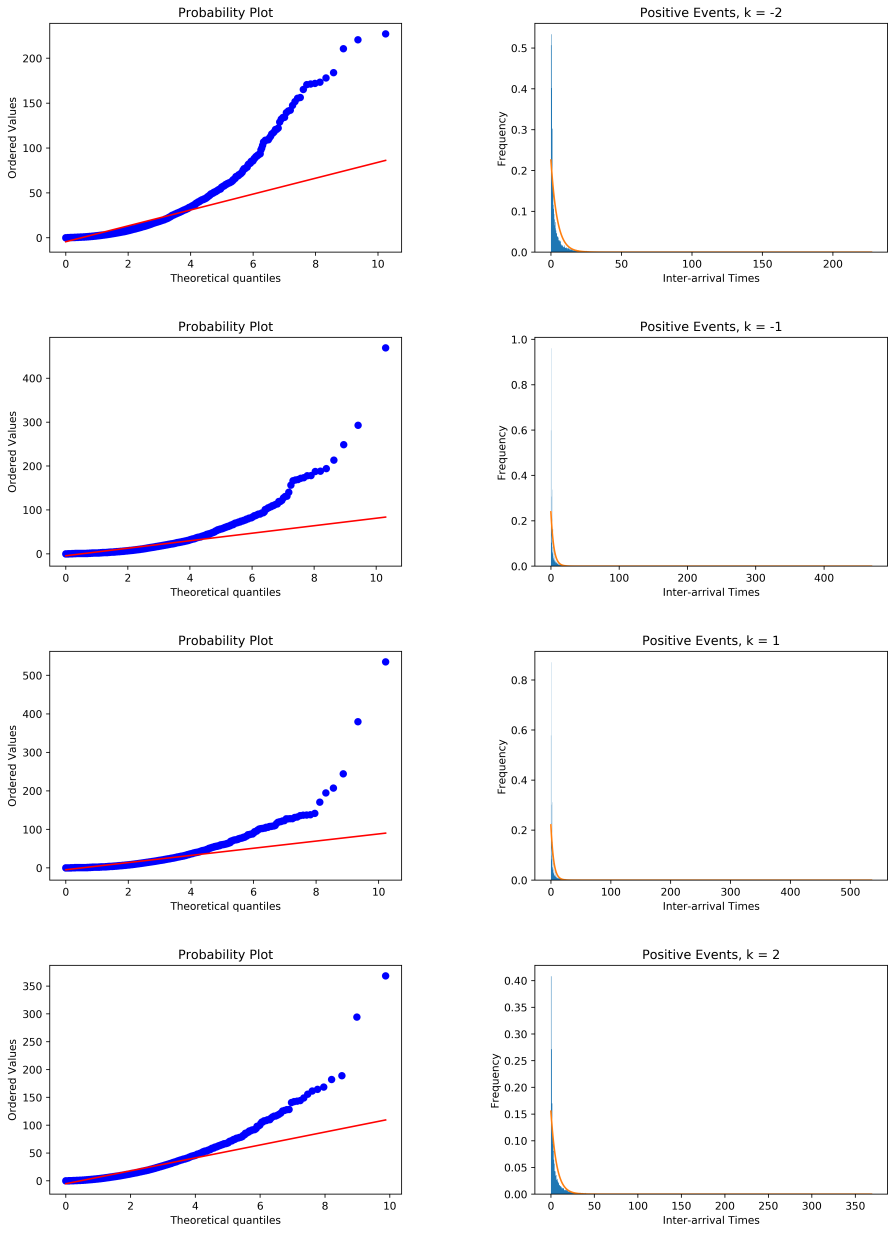
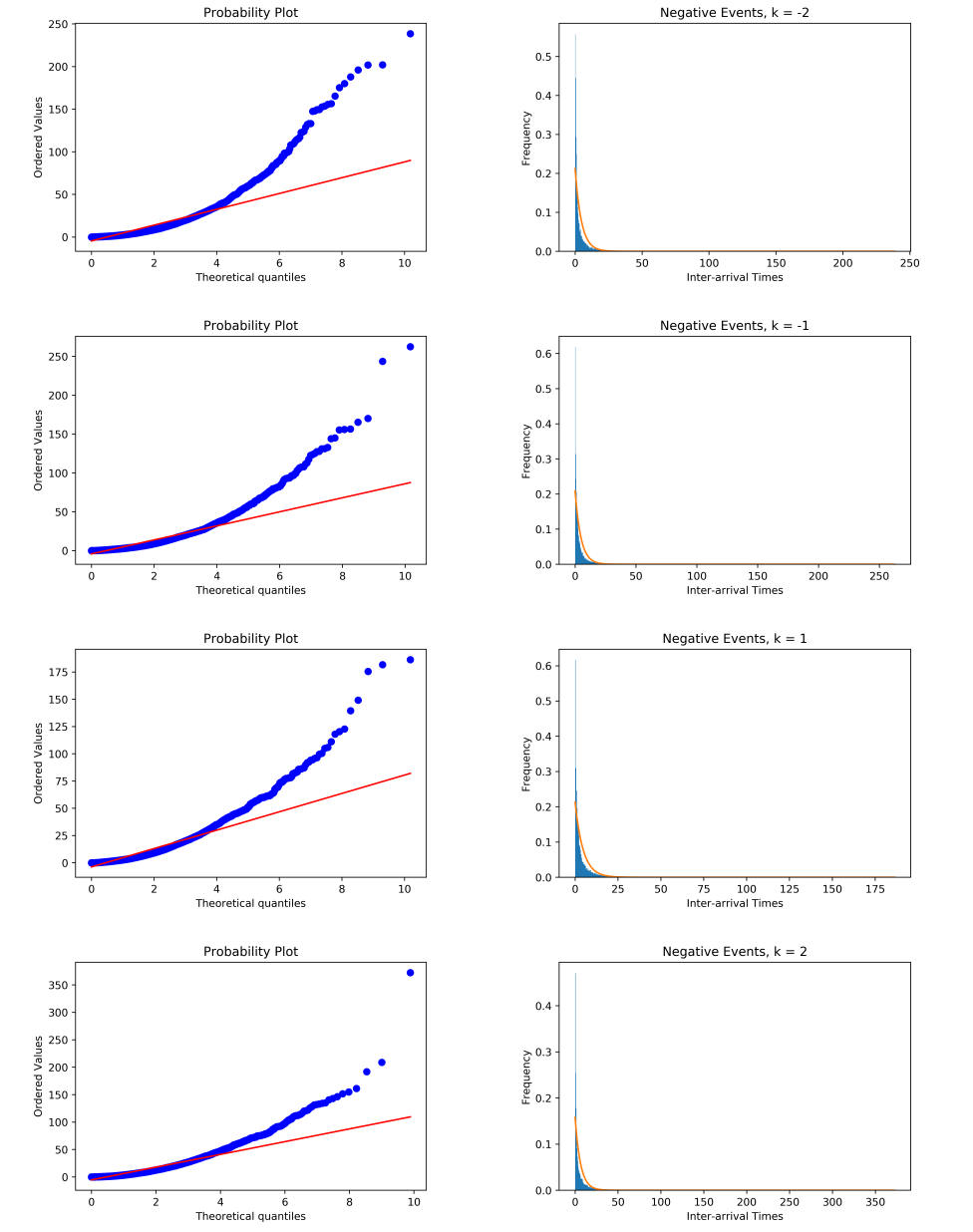


Figure 4.3: Inter-Arrival Times for Negative Events Compared to Exponential Distribution (4 Positions Closest to p_0)



4.5 Arrival Correlations

Although each of the marginal processes can be individually modelled as a Poisson process, there were significant non-zero correlations between the numbers of arrivals at each position. Using a time period t of 60 seconds, the correlations between $N_i^{\pm}(t)$ and $N_j^{\pm}(t)$ are used to generate a correlation matrix R by sampling random intervals of length t throughout the data collection period. With $K = 10$, R is a 40 x 40 matrix and is estimated with 160000 randomly selected intervals. The code written to estimate these correlations is found in Listing A.5.

The correlation matrix for positive events is shown in Figure 4.4 where the entry (i, j) is $\text{corr}(N_i^+(t), N_j^+(t))$. In general, there are high positive correlations between arrivals at a position k and positions $k \pm 1$, and to a lesser extent $k \pm 2$ and $k \pm 3$. The correlations between positions near each other suggest that increased activity near a price incites similar activity in nearby prices. The smaller but significant positive correlations for positions far away from each other could be due to different levels of activity in the market at different times. For example, there may be more trading activity and therefore more events across the board during work hours vs. nighttime hours.

The correlation matrix for positive vs. negative events is shown in Figure 4.5 where the entry (i, j) is $\text{corr}(N_i^+(t), N_j^-(t))$. Particularly notable is the very high (nearly 1) correlation between positive and negative arrivals at the same position. This characteristic could be due to traders reacting to updates by placing orders in the opposite direction, and could indicate that the order book is resilient to changes in the short term.

The correlation matrix for negative events is shown in Figure 4.6 where the entry (i, j) is $\text{corr}(N_i^-(t), N_j^-(t))$. This matrix is similar in nature to the one for positive events, where positions near each other exhibit high correlations and there are lower positive correlations between positions that are further away from each other.

Figure 4.4: Correlation Matrix for $N_i^+(t), N_j^+(t)$, $t = 60$ seconds

	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	1	2	3	4	5	6	7	8	9	10
-10	1.00	0.35	0.25	0.23	0.20	0.16	0.13	0.15	0.06	0.07	0.09	0.07	0.05	0.05	0.00	0.02	0.08	0.08	0.08	0.09
-9	0.35	1.00	0.45	0.31	0.23	0.17	0.12	0.14	0.11	0.14	0.15	0.14	0.10	0.05	0.01	0.02	0.06	0.09	0.16	0.11
-8	0.25	0.45	1.00	0.61	0.44	0.27	0.20	0.23	0.10	0.14	0.16	0.16	0.09	0.04	0.01	0.03	0.07	0.09	0.20	0.09
-7	0.23	0.31	0.61	1.00	0.63	0.31	0.16	0.20	0.04	0.14	0.14	0.16	0.12	0.02	0.02	-0.02	0.03	0.11	0.12	0.06
-6	0.20	0.23	0.44	0.63	1.00	0.63	0.30	0.23	0.04	0.13	0.20	0.21	0.17	0.10	0.00	-0.01	0.04	0.11	0.12	0.07
-5	0.16	0.17	0.27	0.31	0.63	1.00	0.53	0.34	0.14	0.20	0.26	0.22	0.21	0.15	0.09	0.06	0.07	0.09	0.13	0.07
-4	0.13	0.12	0.20	0.16	0.30	0.53	1.00	0.54	0.23	0.23	0.24	0.23	0.23	0.20	0.15	0.14	0.14	0.11	0.16	0.10
-3	0.15	0.14	0.23	0.20	0.23	0.34	0.54	1.00	0.40	0.25	0.24	0.26	0.24	0.21	0.14	0.13	0.19	0.14	0.21	0.11
-2	0.06	0.11	0.10	0.04	0.04	0.14	0.23	0.40	1.00	0.34	0.22	0.21	0.17	0.14	0.13	0.14	0.18	0.12	0.18	0.11
-1	0.07	0.14	0.14	0.14	0.13	0.20	0.23	0.25	0.34	1.00	0.28	0.21	0.20	0.24	0.23	0.18	0.20	0.15	0.18	0.11
1	0.09	0.15	0.16	0.14	0.20	0.26	0.24	0.24	0.22	0.28	1.00	0.38	0.16	0.20	0.21	0.15	0.13	0.10	0.14	0.10
2	0.07	0.14	0.16	0.16	0.21	0.22	0.23	0.26	0.21	0.21	0.38	1.00	0.48	0.21	0.17	0.14	0.14	0.16	0.15	0.09
3	0.05	0.10	0.09	0.12	0.17	0.21	0.23	0.24	0.17	0.20	0.16	0.48	1.00	0.35	0.18	0.11	0.11	0.11	0.14	0.07
4	0.05	0.05	0.04	0.02	0.10	0.15	0.20	0.21	0.14	0.24	0.20	0.21	0.35	1.00	0.28	0.14	0.10	0.06	0.11	0.06
5	0.00	0.01	0.01	0.02	0.00	0.09	0.15	0.14	0.13	0.23	0.21	0.17	0.18	0.28	1.00	0.31	0.16	0.07	0.08	0.06
6	0.02	0.02	0.03	-0.02	-0.01	0.06	0.14	0.13	0.14	0.18	0.15	0.14	0.11	0.14	0.31	1.00	0.30	0.11	0.11	0.09
7	0.08	0.06	0.07	0.03	0.04	0.07	0.14	0.19	0.18	0.20	0.13	0.14	0.11	0.10	0.16	0.30	1.00	0.21	0.17	0.11
8	0.08	0.09	0.09	0.11	0.11	0.09	0.11	0.14	0.12	0.15	0.10	0.16	0.11	0.06	0.07	0.11	0.21	1.00	0.20	0.13
9	0.08	0.16	0.20	0.12	0.12	0.13	0.16	0.21	0.18	0.18	0.14	0.15	0.14	0.11	0.08	0.11	0.17	0.20	1.00	0.33
10	0.09	0.11	0.09	0.06	0.07	0.07	0.10	0.11	0.11	0.11	0.10	0.09	0.07	0.06	0.06	0.09	0.11	0.13	0.33	1.00

These non-zero correlations are clearly non-negligible, so a multivariate Poisson process where the arrivals are correlated and the marginal arrivals follow individual Poisson processes is appropriate. The correlations in Figures 4.4, 4.5, and 4.6 can be combined to form R as described. The procedure for simulating the multivariate Poisson arrival process is discussed in Chapter 5.

Figure 4.5: Correlation Matrix for $N_i^+(t), N_j^-(t)$, $t = 60$ seconds

	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	1	2	3	4	5	6	7	8	9	10
-10	0.68	0.35	0.27	0.22	0.21	0.17	0.15	0.18	0.11	0.12	0.12	0.13	0.09	0.08	0.03	0.05	0.10	0.10	0.12	0.09
-9	0.32	0.85	0.44	0.29	0.22	0.17	0.14	0.14	0.12	0.16	0.13	0.15	0.12	0.08	0.04	0.04	0.07	0.10	0.18	0.11
-8	0.23	0.47	0.91	0.60	0.42	0.25	0.18	0.19	0.09	0.17	0.15	0.17	0.11	0.05	0.01	0.04	0.07	0.10	0.18	0.08
-7	0.23	0.30	0.61	0.96	0.63	0.31	0.15	0.18	0.05	0.16	0.13	0.16	0.13	0.03	0.02	-0.02	0.03	0.12	0.13	0.07
-6	0.19	0.23	0.43	0.62	0.96	0.63	0.29	0.21	0.04	0.14	0.18	0.21	0.18	0.10	0.00	-0.01	0.04	0.11	0.12	0.08
-5	0.16	0.17	0.27	0.32	0.63	0.97	0.55	0.35	0.15	0.22	0.25	0.24	0.23	0.17	0.11	0.08	0.09	0.11	0.15	0.09
-4	0.14	0.14	0.20	0.16	0.29	0.53	0.97	0.54	0.24	0.24	0.23	0.24	0.24	0.21	0.16	0.16	0.15	0.12	0.17	0.11
-3	0.15	0.16	0.24	0.20	0.24	0.35	0.55	0.97	0.42	0.26	0.25	0.27	0.25	0.22	0.14	0.14	0.20	0.16	0.22	0.12
-2	0.09	0.15	0.15	0.10	0.10	0.20	0.30	0.49	0.95	0.39	0.26	0.23	0.20	0.17	0.16	0.17	0.21	0.15	0.21	0.12
-1	0.09	0.13	0.17	0.15	0.17	0.21	0.23	0.25	0.31	0.90	0.30	0.16	0.14	0.20	0.19	0.14	0.14	0.10	0.13	0.09
1	0.05	0.16	0.13	0.10	0.14	0.20	0.19	0.18	0.21	0.31	0.92	0.40	0.18	0.21	0.22	0.16	0.15	0.12	0.15	0.11
2	0.08	0.16	0.17	0.17	0.22	0.24	0.26	0.27	0.22	0.23	0.38	0.96	0.51	0.24	0.19	0.15	0.16	0.17	0.17	0.10
3	0.05	0.10	0.09	0.12	0.17	0.22	0.23	0.25	0.19	0.22	0.17	0.49	0.98	0.37	0.20	0.13	0.13	0.12	0.15	0.08
4	0.04	0.05	0.04	0.03	0.09	0.15	0.21	0.22	0.14	0.24	0.20	0.22	0.36	0.98	0.30	0.16	0.11	0.07	0.12	0.07
5	0.00	0.01	0.01	0.02	0.00	0.09	0.15	0.14	0.12	0.22	0.21	0.16	0.17	0.28	0.98	0.31	0.15	0.07	0.08	0.06
6	0.02	0.02	0.04	-0.01	0.00	0.07	0.14	0.14	0.15	0.18	0.17	0.13	0.11	0.15	0.30	0.95	0.26	0.11	0.10	0.09
7	0.09	0.10	0.09	0.05	0.07	0.09	0.15	0.21	0.18	0.20	0.16	0.14	0.11	0.10	0.14	0.26	0.94	0.20	0.18	0.12
8	0.09	0.10	0.12	0.13	0.14	0.11	0.13	0.17	0.13	0.15	0.13	0.16	0.10	0.05	0.06	0.11	0.20	0.95	0.22	0.15
9	0.10	0.17	0.23	0.15	0.17	0.16	0.18	0.22	0.15	0.14	0.15	0.15	0.13	0.10	0.07	0.09	0.13	0.21	0.80	0.26
10	0.09	0.09	0.07	0.06	0.07	0.07	0.10	0.11	0.10	0.09	0.10	0.08	0.06	0.06	0.06	0.08	0.11	0.12	0.30	0.93

Figure 4.6: Correlation Matrix for $N_i^-(t), N_j^-(t)$, $t = 60$ seconds

	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	1	2	3	4	5	6	7	8	9	10
-10	1.00	0.36	0.27	0.23	0.20	0.18	0.16	0.19	0.14	0.11	0.10	0.14	0.10	0.08	0.02	0.05	0.12	0.11	0.14	0.08
-9	0.36	1.00	0.48	0.28	0.22	0.18	0.16	0.15	0.14	0.12	0.16	0.18	0.12	0.08	0.03	0.04	0.09	0.09	0.17	0.09
-8	0.27	0.48	1.00	0.60	0.42	0.26	0.19	0.21	0.14	0.16	0.14	0.19	0.11	0.05	0.01	0.04	0.08	0.11	0.20	0.07
-7	0.23	0.28	0.60	1.00	0.62	0.32	0.15	0.19	0.10	0.16	0.11	0.18	0.13	0.03	0.02	-0.01	0.05	0.14	0.16	0.06
-6	0.20	0.22	0.42	0.62	1.00	0.64	0.29	0.22	0.08	0.15	0.14	0.23	0.18	0.10	0.00	0.00	0.06	0.13	0.16	0.07
-5	0.18	0.18	0.26	0.32	0.64	1.00	0.55	0.36	0.21	0.21	0.21	0.27	0.24	0.18	0.11	0.08	0.10	0.12	0.17	0.08
-4	0.16	0.16	0.19	0.15	0.29	0.55	1.00	0.54	0.30	0.22	0.20	0.27	0.25	0.22	0.16	0.16	0.15	0.13	0.18	0.11
-3	0.19	0.15	0.21	0.19	0.22	0.36	0.54	1.00	0.51	0.25	0.19	0.29	0.27	0.23	0.14	0.15	0.21	0.18	0.23	0.11
-2	0.14	0.14	0.14	0.10	0.08	0.21	0.30	0.51	1.00	0.37	0.23	0.24	0.22	0.17	0.15	0.18	0.23	0.16	0.19	0.11
-1	0.11	0.12	0.16	0.16	0.15	0.21	0.22	0.25	0.37	1.00	0.26	0.16	0.15	0.19	0.19	0.16	0.17	0.12	0.14	0.09
1	0.10	0.16	0.14	0.11	0.14	0.21	0.20	0.19	0.23	0.26	1.00	0.40	0.19	0.21	0.21	0.16	0.16	0.12	0.13	0.09
2	0.14	0.18	0.19	0.18	0.23	0.27	0.27	0.29	0.24	0.16	0.40	1.00	0.53	0.24	0.17	0.14	0.16	0.16	0.16	0.09
3	0.10	0.12	0.11	0.13	0.18	0.24	0.25	0.27	0.22	0.15	0.19	0.53	1.00	0.38	0.18	0.12	0.12	0.11	0.14	0.07
4	0.08	0.08	0.05	0.03	0.10	0.18	0.22	0.23	0.17	0.19	0.21	0.24	0.38	1.00	0.29	0.16	0.10	0.06	0.10	0.07
5	0.02	0.03	0.01	0.02	0.00	0.11	0.16	0.14	0.15	0.19	0.21	0.17	0.18	0.29	1.00	0.31	0.14	0.06	0.07	0.05
6	0.05	0.04	0.04	-0.01	0.00	0.08	0.16	0.15	0.18	0.16	0.16	0.14	0.12	0.16	0.31	1.00	0.27	0.10	0.09	0.09
7	0.12	0.09	0.08	0.05	0.06	0.10	0.15	0.21	0.23	0.17	0.16	0.16	0.12	0.10	0.14	0.27	1.00	0.21	0.15	0.12
8	0.11	0.09	0.11	0.14	0.13	0.12	0.13	0.18	0.16	0.12	0.12	0.16	0.11	0.06	0.06	0.10	0.21	1.00	0.24	0.14
9	0.14	0.17	0.20	0.16	0.16	0.17	0.18	0.23	0.19	0.14	0.13	0.16	0.14	0.10	0.07	0.09	0.15	0.24	1.00	0.25
10	0.08	0.09	0.07	0.06	0.07	0.08	0.11	0.11	0.11	0.09	0.09	0.09	0.07	0.07	0.05	0.09	0.12	0.14	0.25	1.00

Chapter 5

Simulating the LOB

5.1 Generating Correlated Poisson Variables

In presenting the simulation procedure, it is necessary to discuss the method for generating correlated Poisson variables. The goal is to generate a d-dimensional multivariate random variable $\boldsymbol{X} = (x_1, x_2, \ldots, x_d)$ with mean $(\lambda_1, \lambda_2, \ldots, \lambda_d)$ and d x d correlation matrix R_d where each X_i has a marginal distribution that is the same as the distribution of a Poisson variable with mean λ_i. That is, $P(X_i = x_i) = f_i(x_i)$, where f_i is the probability mass function of a Poisson variable with mean λ_i. Inouye et al. (2017) describe a procedure for generating $\boldsymbol{X}$ using the copula approach.

A copula is defined as any joint probability cumulative distribution function where each marginal distribution is uniformly distributed (Ross (2012)). That is,

$$C(\boldsymbol{u}) : [0,1]^d \to [0,1]$$

is a copula if $C(0, 0, \ldots, 0) = 0$ and $C(1, \ldots, 1, u_i, 1, \ldots, 1) = u_i$ and $u_i \in [0, 1]$ for all i. The Gaussian copula as described in Inouye et al. (2017) is used, where

$$C_R^{Gaussian}(u_1, u_2, \ldots, u_d) = \Phi_{R_d}(\Phi^{-1}(u_1), \Phi^{-1}(u_2), \ldots, \Phi^{-1}(u_d))$$

in which Φ is the cumulative distribution function for a standard random normal variable and Φ_{R_d} is the cumulative distribution function for a multivariate normal random variable with mean $\mathbf{0}$, variance $\mathbf{1}$, and correlation matrix R_d. $\boldsymbol{u}$ can be generated by first generating a random multivariate normal variable $\boldsymbol{Y} = (y_1, y_2, \ldots, y_d)$ with correlation matrix R_d and setting $(u_1, u_2, \ldots, u_d) = (\Phi(y_1), \Phi(y_2), \ldots, \Phi(y_d))$. It can then be paired with a copula that has marginal Poisson distributions. Namely,

$$C^{Poisson}(F_1(x_1), F_2(x_2), \ldots, F_d(x_d))$$

where F_i is the discrete cumulative distribution function for a Poisson random variable with rate λ_i. $\boldsymbol{x}$ can then be set to

$$(x_1, x_2, \ldots, x_d) = (F_1^{-1}(u_1), F_2^{-1}(u_2), \ldots, F_d^{-1}(u_d))$$

To implement $F_i^{-1}(u_i)$, the inverse transform method as described in Ross (2012) is used:

Algorithm 1: Inverse Transform Method for Generating a Poisson Rate Variable With Mean λ and quantile u

Let $i = 0$;

Let $p = e^{-\lambda}$;

Let $F = p$;

while $u \geq F$ **do**

  $p \leftarrow \lambda p/(i+1)$;

  $F \leftarrow F + p$;

  $i \leftarrow i + 1$;

end

return i ;

Each x_i will have a marginal Poisson distribution with rate λ_i and $\boldsymbol{x}$ will approximately have correlation matrix R_d. Table 5.1 shows the accuracy of this method for different desired correlations using simulation. For each desired correlation, 10000 pairs of random variables (x_1, x_2) are generated that have means 1 and joint Poisson distribution (see Listing A.6). For each correlation level, x_1 and x_2 have the correct means that are very near 1. For positive correlations, the actual correlation is slightly below the desired correlation (less than 0.1 difference at each desired correlation). For negative correlations, the actual correlation is slightly above the desired correlation for smaller magnitudes and deviates significantly at higher magnitudes. In practice, our correlation matrix R contains values that range from small negative correlations to large positive correlations (see Tables 4.4, 4.5, and 4.6), so this method would generate Poisson arrivals that follow R closely. It may be possible to generate arrivals that more accurately reflect the given correlations, but this method provides acceptable performance for the given data.

5.2 Backwards Simulation

Using the methods in Section 5.1, the numbers of positive and negative events in a time period of length t can be generated with means equal to $\lambda_k^{\pm} \cdot t$ and correlation matrix R. Given the numbers of arrivals, a method called backwards simulation can be used to simulate the times of events (Bae and Kreinin (2017)).

Backwards simulation uses the fact that conditioned on the number of arrivals in a Poisson process during a time period t being equal to n, these arrivals are distributed uniformly across $[0, t)$. Using this method, arrivals over a larger time period T (i.e. $T = 10t$) can be generated by simulating arrivals in smaller chunks of size t. An algorithm is presented below for simulating arrivals over a time period $[s, s + t)$, with the arrivals following a $4K$-dimensional multivariate Poisson process with means $\lambda_k^{\pm}$

Table 5.1: Actual vs. Desired Correlation Using Gaussian Copula, $n = 10000$

Desired Correlation	$\bar{x_1}$	$\bar{x_2}$	**Actual Correlation**
-1	0.995	1.009	-0.735
-0.9	0.997	0.997	-0.682
-0.8	1.010	0.985	-0.616
-0.7	0.987	1.003	-0.543
-0.6	0.990	1.014	-0.472
-0.5	0.998	1.002	-0.405
-0.4	1.016	0.988	-0.308
-0.3	1.003	1.005	-0.236
-0.2	1.001	1.007	-0.163
-0.1	0.998	1.018	-0.086
0	1.014	0.986	0.016
0.1	0.976	0.992	0.062
0.2	0.994	0.986	0.162
0.3	0.995	1.002	0.246
0.4	1.006	1.004	0.371
0.5	1.000	0.997	0.433
0.6	0.987	1.001	0.522
0.7	1.007	1.007	0.616
0.8	0.993	0.986	0.727
0.9	1.010	1.008	0.829
1	0.997	0.997	1.000

and correlation matrix R. The event sizes are distributed exponentially with means $\mu_k^{\pm}$ as described in Section 4.4. It returns a list of events E whose entries (k, y, τ) consist of the event position, size, and time respectively:

Algorithm 2: Backwards Simulation Method For Generating Correlated Poisson Arrivals During Time Period $[s, s+t)$

$E = []$;

Generate Poisson variables $(n_{-K}^{\pm}, \ldots, n_{-1}^{\pm}, n_1^{\pm}, \ldots, n_K^{\pm})$ with means $(\lambda_{-K}^{\pm}, \ldots, \lambda_{-1}^{\pm}, \lambda_1^{\pm}, \ldots, \lambda_K^{\pm})$ and correlation matrix R (see Section 5.1) ;

for $k = -K, \ldots, -1, 1, \ldots, K$ **do**

  for $i = 1, \ldots, n_k^+$ **do**

    Generate $\tau \sim$ Uniform$(s,\ s+t)$;

    Generate $y \sim$ Exponential(μ_k^+) ;

    Append (k, y, τ) to E ;

  end

  for $i = 1, \ldots, n_k^-$ **do**

    Generate $\tau \sim$ Uniform$(s,\ s+t)$;

    Generate $y \sim$ Exponential(μ_k^-) ;

    Append $(k, -y, \tau)$ to E ;

  end

end

Return E ;

Sample arrivals using parameters estimated in Chapter 4 are shown in Figures 5.1 and 5.2. Positive events are shown in blue while negative events are shown in orange.

Figure 5.1: Bid Arrivals over $T = 600$ seconds

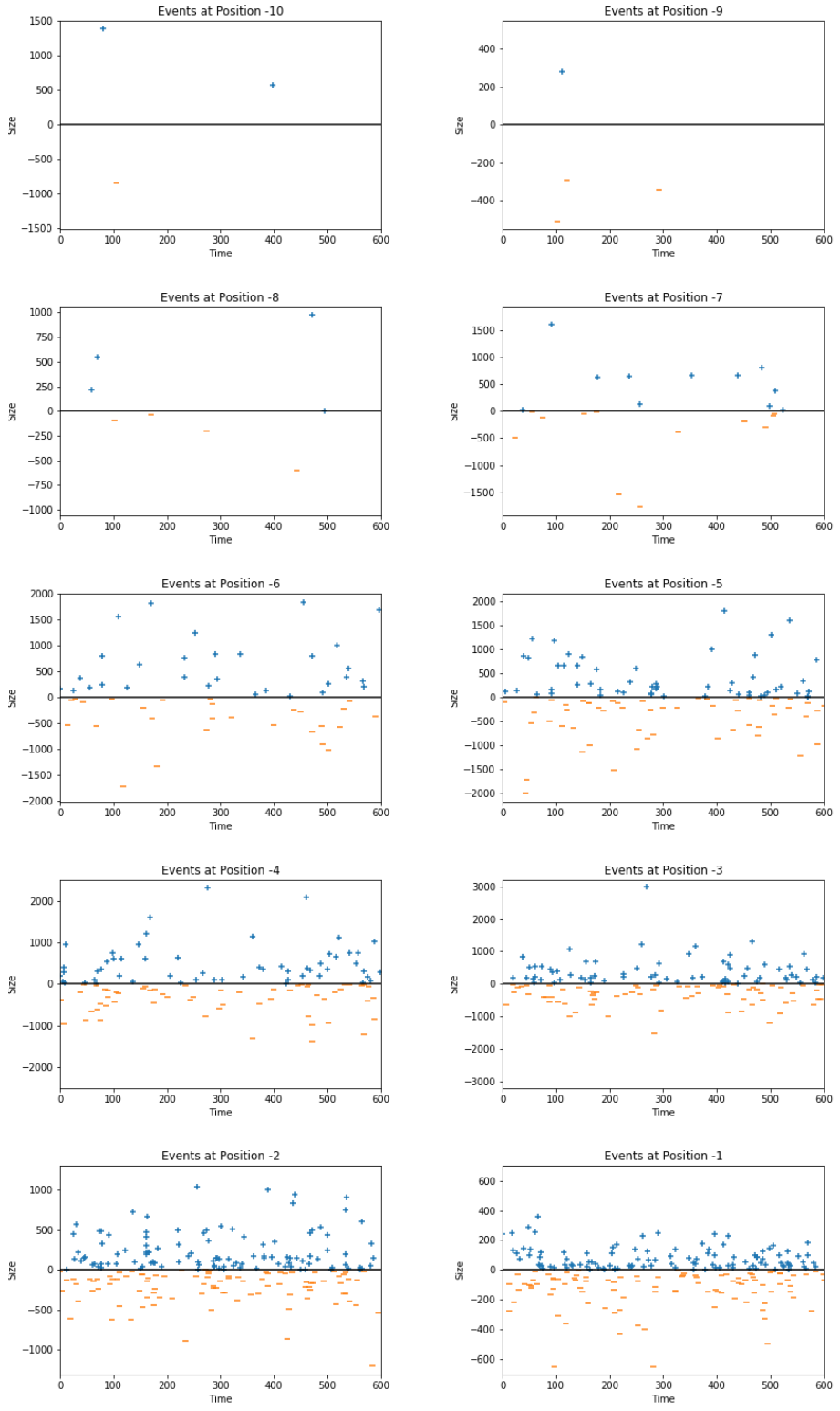
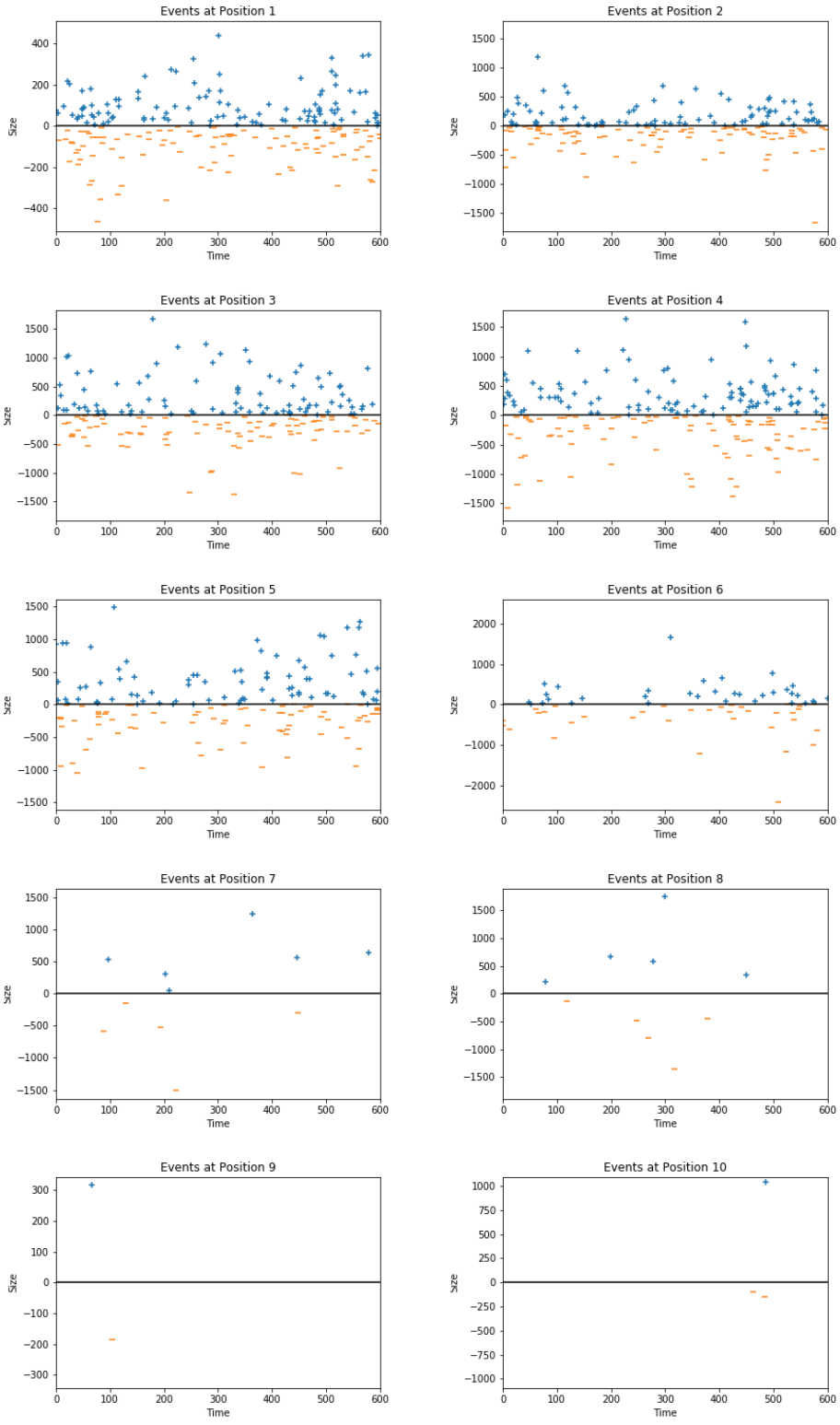


Figure 5.2: Ask Arrivals over $T = 600$ seconds



5.3 Simulation Algorithm

The actions of the agent can now be incorporated into the simulation. The agent must buy V units in a time period T. It follows a trading schedule Ω, which is a list of market orders of the form (M, τ) where M is the size of the order and τ is the time at which it is to be sent. If V units have not been acquired at the end of the time period, the agent makes a market order for the remainder of the inventory to comply with the spirit of the game. LOB events outside of the agent's orders are generated using Algorithm 2, and the reference price is updated as described in Section 4.1. The simulation algorithm is presented below. It returns Θ, a list of executed trades of the form (p, a, τ) that contains the price, number of units, and time of the trade. The prices are given assuming that there are no trading fees in the exchange.

Algorithm 3: LOB Simulation: Setup and Input

Let p_0 be the starting reference price ;

Let L be the initial LOB. Let L_p be the volume at price p ;

Let v be the amount of inventory filled ;

Let $\Theta = []$ be the trades executed ;

Let Ω be the list of market orders in the trading schedule. If $\sum_{\Omega} M < V$,

append $(V - \sum_{\Omega} M, T)$ to Ω ;

Generate E using Algorithm 2 ;

Sort Ω and E by τ;

Algorithm 4: LOB Simulation: Recording Executed Trades

Proceed through Ω and E in time order.

while $\tau <= T$ *and* $v < V$ **do**

  if event of form (k, c, τ) **then**

    Update L and p_0 (see Section 4.1) ;

  end

  if order of form (M, τ) **then**

    Let $p = p_0 + 0.5$;

    while $L_p = 0$ **do**
      $p \leftarrow p + 1$

    end

    Let $m = M$;

    while $m > 0$ **do**

      Let $a = (L_p - m)^+$;

      $L_p \leftarrow L_p - a$;

      Append (p, a, τ) to Θ ;

      $m \leftarrow m - a$;

      $p \leftarrow p + 1$;

    end

    $v \leftarrow v + M$;

    Update p_0 ;

  end

end

Return Θ ;

Θ can then be used to compute the VWAP of the trading schedule. See Listing A.7 for the code written to perform the simulation.

Chapter 6

Evaluating Trade Execution Performance

The simulated market environment can be used to assess the performance of a given trade execution strategy. Using this procedure, the agent can adjust its strategy to acquire its inventory goal at the most favorable price before executing the trade in reality. Section 6.1 provides an overview of common algorithmic trade execution strategies. Section 6.2 evaluates the performance of the TWAP and VWAP strategies (see Subsection 6.1.1) in the simulated market environment.

6.1 Trade Execution Strategies

Trade execution strategies can be broadly classified into impact-driven, cost-driven, and opportunistic algorithms (Labadie and Lehalle (2010)). Nuanced trading strategies can be created using a combination of these algorithms.

6.1.1 Impact-Driven Algorithms

Impact-driven algorithms seek to minimize the market impact of a large trade by splitting it into slices of smaller trades over time.

Time Weighted Average Price (TWAP) algorithms split a large order of size V into n smaller orders of size V/n uniformly over a time period T. In order to reduce predictability and prevent detection by third parties, modifications of TWAP where the size and timing of orders are slightly varied are often preferred. When simulating TWAP performance, the default TWAP strategy is used, assuming that no third party takes action upon detecting our TWAP strategy and that the market reacts as it normally would to our trades.

Volume Weighted Average Price (VWAP) algorithms are similar to TWAP algorithms, but they divide the time period T into sub-periods where the volume of market activity measurably differs. The size of the orders during each sub-period is proportional to its volume of market activity during that sub-period. The volume of market activity in each sub-period is typically predicted using historical data. VWAP strategies theoretically lower market impact compared to TWAP strategies by sending larger orders when there is more market activity (and thus higher LOB resiliency) and smaller orders when there is less market activity. However, they run the risk of incurring significantly higher execution costs if the historical data does not reflect market activity at the time of execution. The VWAP trading strategy should not be confused with the VWAP benchmark used to measure the performance of trade execution (Chapter 1).

Percentage of Volume algorithms (POV), like VWAP, generate sizes of orders based on trading volume throughout the time period. However, they trade a fixed percentage of the current market volume based on a desired participation rate. For example, the agent may choose to submit orders so that are expected to comprise 5% of the market activity in the period. Unlike the TWAP and VWAP algorithms that

follow fixed trading schedules, POV algorithms have dynamically determined trading schedules.

6.1.2 Cost-Driven Algorithms

Cost-driven algorithms seek to minimize transaction costs, including market impact and market timing risks. In addition to market impact consideration taken in the impact-driven algorithms above, cost-driven algorithms adjust the time horizon of trades to account for market timing risks (i.e. orders for more volatile assets are executed in a shorter time horizon). Implementation Shortfall algorithms attempt to minimize the difference between the price at which the investor decides to make the trade and the average price at which the trade is executed. Market Close algorithms are similar to Implementation Shortfall algorithms, but the benchmark of comparison is the closing price.

6.1.3 Opportunistic Algorithms

Opportunistic algorithms seek to take advantage of favorable market conditions. Price Inline algorithms dynamically adjust trading patterns in response to the asset price (i.e. increasing trading activity when the price is low). Liquidity Driven algorithms take into account order book depth and availability of venues for trading. Pair Trading algorithms consist of buying one asset and selling another. If the assets are sufficiently correlated, the risks from one asset balance out the risks from the other. Pair Trading takes advantage of mean-reverting behavior to generate profit.

6.2 Trade Execution Simulation

In this section, the performance of TWAP and VWAP strategies are tested in the simulated market environment.

6.2.1 The Initial LOB

To begin with an LOB that is representative of the market, the reference price is set to the time weighted average price of the asset rounded to the nearest mid-price. The volumes at the first $K = 10$ surrounding bid and ask prices are set to the time weighted average volumes at those positions, and the volumes at other positions are set to 100. The LOB is maintained as simulation proceeds. The starting book is shown in Table 6.1.

Table 6.1: Starting LOB for Simulation. $p_0 = 516.5$ cents

Bids		**Asks**	
Volume	Price	Price	Volume
587.93	516	517	495.79
1425.68	515	518	983.24
2328.27	514	519	1475.59
2735.44	513	520	2170.60
2578.26	512	521	2456.86
1338.90	511	522	1807.16
609.82	510	523	817.16
292.86	509	524	420.61
250.14	508	525	404.11
287.13	507	526	390.99

6.2.2 TWAP Strategy

As described in Subsection 6.1.1, the TWAP strategy consists of splitting the large trade into equally-sized smaller trades uniformly across the time period. The TWAP strategy is tested with a total trade size of $V = 40000$ over a period of $T = 10$ minutes and different numbers of orders to split the trade into. An example is shown in Table 6.2 where the trade is split into 8 orders of size 5000. As can be seen, the trade pushes the market price up while it is executed. The best ask price at the time of the first order is 517 and the best ask price at the time of the last order is 528 for a

VWAP price of 524.820, about 8 cents higher than the starting reference price. See Listing A.8.1 for the code written to simulate the orders resulting from a trade using the TWAP strategy.

The TWAP strategy was then tested with different numbers of orders in the trading schedule. Each trading schedule was tested 10000 times with the same starting LOB. The results are shown in Table 6.3.

As can be seen for this particular trade size, splitting the trade into a higher number of orders significantly lowers the mean cost up to about 8 orders. Looking at the standard errors, the variability of performance generally decreases as the number of orders increases (with the exception of using just one order). Although increasing the number of orders reduces average trading cost and risk, it has drawbacks in that it may incur extra costs from exchange commissions and is more easily detectable. These simulation results are useful for an agent looking to find the optimal number of slices to use in its TWAP strategy.

6.2.3 VWAP Strategy

As described in 6.1.1, the VWAP strategy consists of splitting trades like the TWAP strategy, but determining order sizes in proportion to the volume of market trades during different sub-periods. To simulate an environment with varying levels of trading activity, the 10 minute period is split in half, where a certain proportion α $(0 < \alpha < 1)$ of the activity occurs in the first half and $1 - \alpha$ of the activity occurs in the second half.

In order to maintain the same average volume of trades over the entire period, the rate of arrival λ_k at a given position is adjusted as shown in Table 6.4.

The expected number of arrivals in the first half will be

$$2\alpha\lambda_k T/2 = \alpha\lambda_k T$$

Table 6.2: TWAP Strategy Order Example

VWAP Price: 524.820			
Time	**Order Size**	**Price**	**Volume**
66.67	5000	517	682.696
		518	428.594
		519	2566.022
		520	1301.644
		521	21.044
133.33	5000	520	260.565
		521	1454.826
		522	2239.506
		523	490.201
		524	554.901
200.00	5000	522	62.550
		523	133.966
		524	3165.501
		525	1637.983
266.67	5000	521	302.563
		522	411.777
		523	584.960
		524	3541.595
		525	159.105
333.33	5000	524	684.054
		525	981.617
		526	3334.330
400.00	5000	525	891.352
		526	1403.401
		527	1557.507
		528	1147.739
466.67	5000	526	631.725
		527	2759.918
		528	1236.733
		529	371.624
533.33	5000	528	461.628
		529	583.475
		531	3954.897

Table 6.3: TWAP Strategy Performance

No. Orders	**Mean VWAP Price (Standard Error)**
1	573.481 (0.084)
2	566.053 (0.116)
3	558.112 (0.130)
4	551.222 (0.139)
5	542.895 (0.136)
6	536.191 (0.120)
7	531.526 (0.096)
8	529.286 (0.081)
9	529.482 (0.082)
10	528.490 (0.070)
100	529.318 (0.061)

Table 6.4: Adjusted Rates for VWAP Strategy Simulation

Subperiod	**Adjusted Rate**
First Half	$2\alpha\lambda_k$
Second Half	$2(1-\alpha)\lambda_k$

and the expected number of arrivals in the second half will be

$$2(1-\alpha)\lambda_k T/2 = (1-\alpha)\lambda_k T$$

for a total of

$$\lambda_k T$$

Thus, on average, α proportion of arrivals will occur in the first half and $1-\alpha$ proportion of arrivals occur in the second half with the same average total number of arrivals as the constant rate case. The size of the orders will be adjusted in the same way. It should be noted that these trading conditions are somewhat unrealistic. Although it is possible that trading volume could differ significantly in a short time period, it is unlikely that the agent would be able to accurately and uniquely predict such a drastic change and act upon it. Nevertheless, this analysis shows the possibility

of testing the performance of a trading strategy in a simulated environment with varying rates of market activity. Table 6.5 shows an example of the orders executed with a VWAP strategy split into 8 slices when $\alpha = 0.75$, where the size of the orders in the first half is 7500 and the size of the orders in the second half is 2500. Again, the price is pushed up. The best ask price at the time of the last order is 528 and the VWAP price is 522.507, about 6 cents higher than the starting reference price. See Listing A.8.2 for the code written to simulate the orders resulting from trades using the VWAP strategy.

Table 6.6 shows the performance of the VWAP strategy (volume-adjusted trading) vs. the TWAP strategy (no volume-adjusted trading) for various α and numbers of orders.

It can be seen that for low numbers of orders (e.g. 4 orders), the TWAP strategy significantly outperforms the VWAP strategy, and this difference becomes more pronounced as α increases. This difference is likely due to the fact that the starting order book is the same in each case. Since orders in the first half of the VWAP strategy are larger, they reach farther into the order book and more units are bought at higher prices. Besides reaching farther into the order book, these orders have the additional market impact effect of increasing the price more in the first half. Thus, prices in the second half are higher even with lower trading activity.

The difference is mitigated when the number of orders is increased since the smaller orders do not reach as far into the order book at the beginning of the trading period. However, the TWAP strategy still largely outperforms the VWAP strategy for $\alpha = 0.6$ and $\alpha = 0.7$, likely for similar reasons as described above. When both α and the numbers of orders are high (i.e. $\alpha = 0.8$ with 16 or 20 orders, $\alpha = 0.9$ with 12, 16, or 20 orders), the VWAP strategy outperforms the TWAP strategy. These cases are ideal for the VWAP strategy, since higher market activity in the first half means that the LOB recovers more quickly from the larger order sizes and the market impact

Table 6.5: VWAP Strategy Order Example, $\alpha = 0.75$

VWAP Price: 522.507			
Time	**Order Size**	**Price**	**Volume**
66.67	7500	517	1071.610
		519	1907.321
		520	4116.339
		521	404.731
133.33	7500	518	2145.428
		521	4555.106
		522	799.466
200.00	7500	518	207.043
		519	856.150
		520	1513.202
		521	835.556
		522	1043.699
		523	3044.349
266.67	7500	521	507.424
		522	2825.699
		523	1860.746
		524	1134.190
		525	1171.941
333.33	2500	524	516.300
		525	1676.603
		526	307.097
400.00	2500	525	968.908
		527	759.657
		528	771.434
466.67	2500	525	357.558
		526	71.188
		527	807.992
		528	1263.263
533.33	2500	528	639.090
		529	117.624
		530	1743.286

is decreased. Although these conditions do not completely reflect a realistic market environment, these results show that an ideal VWAP strategy should take into account the market activity, number of orders, and depth of the LOB.

Table 6.6: VWAP Strategy Performance

	Mean VWAP Price (Standard Error)	
No. Orders	**Volume-Adjusted Trading**	**No Volume-Adjusted Trading**
$\boldsymbol{\alpha = 0.6}$		
4	547.869 (0.156)	545.642 (0.144)
8	529.340 (0.085)	528.208 (0.071)
12	528.065 (0.068)	527.514 (0.059)
16	527.674 (0.059)	527.383 (0.052)
20	527.840 (0.057)	527.366 (0.050)
$\boldsymbol{\alpha = 0.7}$		
4	547.437 (0.158)	541.349 (0.142)
8	529.788 (0.093)	528.257 (0.067)
12	528.086 (0.072)	527.576 (0.059)
16	527.692 (0.063)	527.492 (0.055)
20	527.500 (0.057)	527.513 (0.052)
$\boldsymbol{\alpha = 0.8}$		
4	548.495 (0.152)	537.137 (0.133)
8	530.943 (0.105)	529.014 (0.072)
12	528.423 (0.076)	528.485 (0.063)
16	527.707 (0.066)	528.463 (0.064)
20	527.502 (0.059)	528.692 (0.063)
$\boldsymbol{\alpha = 0.9}$		
4	550.013 (0.148)	534.170 (0.124)
8	531.776 (0.110)	530.629 (0.077)
12	528.574 (0.080)	530.484 (0.076)
16	527.674 (0.066)	530.641 (0.075)
20	527.373 (0.060)	530.816 (0.076)

Chapter 7

Conclusion

7.1 Summary of Results

In this thesis, a modified version of the queuing model developed by Huang et al. (2013) was used to simulate LOB dynamics. Data from the ETC-USD market of the Coinbase Pro Exchange was used to develop the model and estimate parameters. Several aspects of the original model were validated from the data set, such as the fact that arrivals at individual positions marginally follow Poisson processes. Other aspects were adjusted in the new model to better fit the observed data. Positive and negative arrivals of events at different bid and ask positions were found to be significantly correlated, so they were jointly modelled as a multivariate Poisson process. The sizes of these events were modelled as exponential random variables.

The model was used to artificially simulate the behavior of an LOB over time. This simulated market environment was used to evaluate the performance of TWAP and VWAP trade execution strategies when acquiring a large amount of inventory in a short period of time, with the agent's trades impacting the LOB dynamics in real time. The performance of TWAP strategies improved as the number of orders increased, with diminishing returns after a certain point. The VWAP strategy outperformed

the TWAP strategy when the volume activity in the market varied substantially throughout the time period and the number of orders was sufficiently large.

7.2 Future Research

7.2.1 Improving the Queuing Model

Several improvements could possibly make the queuing model more accurately reflect real-life LOB dynamics.

As in the model used by Huang et al. (2013), negative arrivals could be split into cancellation orders and market orders where their arrival rates are estimated separately. This differentiation would eliminate the need for the part of the model where negative events at the best bid or ask price are assumed to be market orders and negative events elsewhere are assumed to be limit order cancellations. Unfortunately, because Coinbase Pro does not specify the types of its negative updates, this differentiation is not possible with our data set, but it may be possible using a different data set.

The model also assumes that arrivals at each position are independent of the current state of the LOB. In Huang et al. (2013), for example, the arrival rates differ based on the depth of the LOB at the given price. Rates are also affected by whether the price is the best bid or ask price. Although different LOB states likely affect arrival behavior, there was not enough data collected to estimate the rates given different LOB states. More data is needed to consistently estimate these parameters. The same can be said about the size of arrivals, which may not be independent of the LOB state.

The arrival rate and average event size estimates mean that the LOB evolves over time purely from endogenous activity (besides the agent's activity). Exogenous effects could also be introduced in the model if the price of the asset is expected to drift up

or down over time (e.g. from a price shock).

7.2.2 Further Trading Execution Strategy Testing

In Section 6.2, the performance of two simple, widely used strategies TWAP and VWAP were tested. Different strategies such as the ones described in Section 6.1 could also be evaluated in the simulated environment. The simulation program at the moment only allows for trading schedules that contain just market orders, as it is the only type of order needed for the TWAP and VWAP strategies. It could be expanded to allow for additional agent activities such as limit orders, stop-loss orders, and order cancellations that are used in more advanced strategies.

Since the model adjusts LOB dynamics based on agent actions, it could be used to expand upon the studies performed Nevmyvaka et al. (2006) and Spooner et al. (2018) that use reinforcement learning to tackle optimal trade execution problems. Reinforcement learning could be applied and tested within the simulated market environment.

The model could also incorporate multiple players with different behaviors. For example, trade execution strategies could be tested in the presence of an antagonistic player that reacts to an agent's actions or with multiple players using different trading strategies.

7.2.3 Applying the Model to Different Securities

Finally, the work in this thesis is broadly applicable to modelling the behavior of LOBs for securities across different exchanges. High frequency LOB data is available for many exchanges that broker trades of stocks, options, futures, and other securities. For example, `https://lobsterdata.com` offers high frequency LOB data for equities on the NASDAQ exchange. Data like LOBSTER could be used to model the behavior of different securities' LOBs. Trade execution strategies could

be tested using the simulation methods from this thesis with adjustments made to reflect different characteristics and trading rules of the exchange.

Appendix A

Code Listings

This appendix contains code written for several important sections of this thesis. With this code, the reader should be able to reproduce the experiment and run his or her own analysis. The full code used in this thesis can be found here: `https://github.com/aw21/Thesis`.

A.1 Software Dependencies

Listing A.1: Software Requirements

```
import asyncio
from copra.websocket import Channel, Client
import matplotlib.pyplot as plt
from collections import OrderedDict
from time import sleep
from dateutil import parser
import copy
import datetime
import itertools
from operator import itemgetter
import numpy as np
from dateutil.tz import tzutc
import math
import pytz
from pytz import timezone
import pickle
import pylab
from scipy.stats import probplot,expon,kstest,norm
import matplotlib.pyplot as plt
```

```
import json
from datetime import timedelta
import pandas as pd
from scipy.stats.stats import pearsonr
import bisect
import scipy.stats as ss
import bisect
from mpl_toolkits.mplot3d import Axes3D
```

A.2 Raw Data Collection

Listing A.2: Receiving LOB Data From Coinbase Pro

```
class Level2(Client):
    global file
    num_updates = 0

    # Receives message from API websocket
    def on_message(self, message):
        # Get snapshot of LOB from API and build internal representation
        if message['type'] == 'snapshot':
            self.starting_time = datetime.datetime.now(datetime.timezone
                ↪ .utc)
            print('Starting time: {}'.format(self.starting_time))
            print()
            json.dump([str(datetime.datetime.now(datetime.timezone.utc))
                ↪ , message['bids'], message['asks']],file)
            file.write("\n")


        # Update order book when new event occurs
        if message['type'] == 'l2update' and 'time' in message:
            for (side, price, amount) in message['changes']:
                json.dump({
                    "side": side, \
                    "price": price, \
                    "amount": amount, \
                    "time": message['time']
                },file)
                file.write("\n")
                self.num_updates += 1
                if self.num_updates % 1000 == 0:
                    print("Number of updates: {}".format(self.
                        ↪ num_updates))
                    print("Most Recent Update: {}".format((side, price,
                        ↪ amount)))
                    print("Time: {}".format(message['time']))
                    print()

    def on_error(message, reason):
        super().on_error(message,reason)
```

```
        print("Error: {}".format(message))


file = open("1_4_19_Long.json","w")
loop = asyncio.get_event_loop()
channel = Channel('level2', 'ETC-USD')
ws = Level2(loop, channel)

async def my_task(seconds):
    global loop
    print('Collecting data for {} seconds'.format(seconds))
    await asyncio.sleep(seconds)
    await ws.close()
    return "Finished Task"

try:
    task_obj = loop.create_task(my_task(seconds=86400))
    loop.run_until_complete(task_obj)
except Exception as e:
    print(e)
finally:
    loop.close()
    file.close()
    print("Finished collecting data")
```

A.3 Raw Data Processing

Listing A.3: Building The Shortened LOB

```
def parse_file_json(file_name):
    with open(file_name, "r") as f:
        data = [json.loads(line) for line in f]
    starting_time = parser.parse(data[0][0])
    starting_bids = data[0][1]
    starting_asks = data[0][2]

    raw_updates = data[1:]

    Bids = {}
    Asks = {}

    updates = []

    for price, amount in starting_bids:
        Bids[int(round((float(price)*100)))] = float(amount)
    for price, amount in starting_asks:
        Asks[int(round((float(price)*100)))] = float(amount)

    for u in raw_updates:
        price = int(round((float(u["price"])*100)))
        side = u["side"]
```

```
            amount = float(u["amount"])
            time = parser.parse(u["time"])

            if side == "buy":
                change = amount - Bids.get(price,0)
            else:
                change = amount - Asks.get(price,0)

            updates.append({\
                "Bids": copy.copy(Bids), \
                "Asks": copy.copy(Asks), \
                "time": time, \
                "side": side, \
                "price": price, \
                "change": change
            })

            if side == "buy":
                if u["amount"] == "0":
                    del Bids[price]
                else:
                    Bids[price] = float(amount)
            else:
                if u["amount"] == "0":
                    del Asks[price]
                else:
                    Asks[price] = float(amount)

    return(starting_time, updates)

def shortened_updates(file_name,K):
    starting_time, updates = parse_file_json(file_name)

    res = []

    # Caluclate first reference price
    starting_bids = updates[0]['Bids']
    starting_asks = updates[0]['Asks']
    sorted_bids = list(reversed(sorted(starting_bids.items())))
    sorted_asks = list(sorted(starting_asks.items()))
    best_bid = sorted_bids[0][0]
    best_ask = sorted_asks[0][0]
    if ((best_bid + best_ask) % 2) != 0:
        old_reference_price = round((best_bid+best_ask)/2, 1)
    else:
        old_reference_price = round((best_bid+best_ask+1)/2, 1)

    for u in updates:
        # Find reference price
        sorted_bids = list(reversed(sorted(u['Bids'].items())))
        sorted_asks = list(sorted(u['Asks'].items()))
        best_bid = sorted_bids[0][0]
        best_ask = sorted_asks[0][0]
        if ((best_bid + best_ask) % 2) != 0:
```

```
            reference_price = round((best_bid+best_ask)/2, 1)
        else:
            middle = (best_bid+best_ask)/2
            if old_reference_price > middle:
                reference_price = round((best_bid+best_ask)/2 + 0.5,1)
            else:
                old_reference_price = round((best_bid+best_ask)/2 -
                    ↪ 0.5,1)

        shortened_book = OrderedDict([(k,0) for k in range(-K,K+1) if k
            ↪ != 0])
        first_bid = int(round(reference_price - 0.5))
        first_ask = int(round(reference_price + 0.5))
        for k in range(-K,0):
            shortened_book[k] = u['Bids'].get(first_bid + k + 1,0)
        for k in range(1,K+1):
            shortened_book[k] = u['Asks'].get(first_ask + k - 1,0)

        # Find k from the price. Keep track of event if
        # abs(k) <= K
        price = u["price"]
        k = price - reference_price
        if k < 0:
            k = int(round(k - 0.5))
        else:
            k = int(round(k + 0.5))
        if abs(k) <= K:
            res.append({
                'reference_price': reference_price,
                'LOB': copy.copy(shortened_book),
                'k': k,
                'change': u['change'],
                'time': u['time']
            })

        old_reference_price = reference_price

    return starting_time, res

def processed_updates(file_name,K):
    starting_time,updates = shortened_updates(file_name,10)
    grouped_by_time = OrderedDict([(k, list(v)) for k, v in itertools.
        ↪ groupby(updates, key=lambda x:x['time'])])
    # Contains dictionary of time, reference price, order book, list of
        ↪ changes
    cleaned_updates = []
    for t, us in grouped_by_time.items():
        if len(us) == 1:
            u = us[0]
            cleaned_updates.append(copy.copy(u))
        else:
            new_update = {'time': t}
            us = sorted(us, key=lambda u:-abs(u['k']))
            grouped_by_k = OrderedDict((k, list(v)) for k, v in
```

```
                ↪ itertools.groupby(us, key=lambda u:u['k']))
            reference_k = list(grouped_by_k.keys())[0]
            new_update['reference_price'] = grouped_by_k[reference_k
                ↪ ][0]['reference_price']
            new_update['LOB'] = copy.copy(grouped_by_k[reference_k][0]['
                ↪ LOB'])
            events = []
            for k in grouped_by_k:
                combined_change = 0
                for u in grouped_by_k[k]:
                    combined_change = combined_change + u['change']
                events.append((k,combined_change))
            for k,change in events:
                new_update = copy.deepcopy(new_update)
                new_update['k'] = k
                new_update['change'] = change
                cleaned_updates.append(new_update)

    combined_updates = []
    i = 0
    while i < len(cleaned_updates):
        reference_price = cleaned_updates[i]['reference_price']
        j = i
        updates_at_reference = []
        while (j < len(cleaned_updates)) and (cleaned_updates[j]['
            ↪ reference_price'] == reference_price):
            updates_at_reference.append(cleaned_updates[j])
            j += 1
        updates_at_reference = sorted(updates_at_reference,key=lambda u:
            ↪ u['k'])
        grouped_by_k = OrderedDict((k, list(v)) for k, v in itertools.
            ↪ groupby(updates_at_reference, key=lambda u:u['k']))
        for k,us in grouped_by_k.items():
            us = copy.deepcopy(sorted(us,key=lambda u:u['time']))
            keep_index = [True for u in us]
            for m in reversed(range(1,len(us))):
                quick_same_order = (us[m]['time'] - us[m-1]['time']).
                    ↪ total_seconds() < 0.01
                same_sign = (us[m]['change'] * us[m-1]['change']) > 0
                if quick_same_order and same_sign:
                    us[m-1]['change'] += us[m]['change']
                    keep_index[m] = False
            for (u,keep) in zip(us, keep_index):
                if keep:
                    combined_updates.append(u)
        i = j + 1

    combined_updates = sorted(combined_updates,key=lambda u:u['time'])

    ending_time = combined_updates[-1]['time']

    return starting_time, ending_time, combined_updates

starting_times, ending_times, time_period_updates = zip(*[
```

```
    ↪ processed_updates(f,10) for f in ['12_28_18.json','12_29_18.json',
    ↪ '12_30_18.json']])
```

A.4 Calculating Average Event Sizes and Arrival Rates

Listing A.4: Finding Average Event Sizes And Arrival Rates

```
K = 10
def AES_and_Rate(updates, starting_times):
    event_sizes_pos = OrderedDict([(k,[]) for k in range(-K,K+1) if k !=
        ↪  0])
    event_sizes_neg = OrderedDict([(k,[]) for k in range(-K,K+1) if k !=
        ↪  0])

    for updates in all_updates:
        for u in updates:
            if u['change'] > 0:
                event_sizes_pos[u['k']].append(u['change'])
            else:
                event_sizes_neg[u['k']].append(-u['change'])

    total_time = np.sum([(e-s).total_seconds() for s,e in zip(
        ↪ starting_times,ending_times)])

    AESs = OrderedDict([((k,"pos"),0) for k in range(-K,K+1) if k != 0]
        ↪ + \
                       [((k,"neg"),0) for k in range(-K,K+1) if k != 0])
    numbers_of_events = OrderedDict([((k,"pos"),0) for k in range(-K,K
        ↪ +1) if k != 0] + \
                                    [((k,"neg"),0) for k in range(-K,K+1)
                                        ↪  if k != 0])
    average_rates = OrderedDict([((k,"pos"),0) for k in range(-K,K+1) if
        ↪  k != 0] + \
                                [((k,"neg"),0) for k in range(-K,K+1) if
                                    ↪  k != 0])

    for (k,sizes) in event_sizes_pos.items():
        AESs[(k,"pos")] = np.mean(sizes)
        numbers_of_events[(k,"pos")] = len(sizes)

    for (k,sizes) in event_sizes_neg.items():
        AESs[(k,"neg")] = np.mean(sizes)
        numbers_of_events[(k,"neg")] = len(sizes)

    for k in numbers_of_events:
        average_rates[k] = numbers_of_events[k] / total_time

    return AESs, numbers_of_events, average_rates
```

```
AESs, numbers_of_events, average_rates = AES_and_Rate(all_updates,
    ↪ starting_times)

for k in range(-K,K+1):
    if k != 0:
        print("k = {}".format(k))
        print("AES Positive: {}".format(AESs[(k,"pos")]))
        print("n = {}".format(numbers_of_events[(k,"pos")]))
        print("Average Rate: {}".format(average_rates[(k,"pos")]))
        print("Multiplied: {}".format(AESs[(k,"pos")]*average_rates[(k,"
            ↪ pos")]))
        print("AES Negative: {}".format(AESs[(k,"neg")]))
        print("n = {}".format(numbers_of_events[(k,"neg")]))
        print("Average Rate: {}".format(average_rates[(k,"neg")]))
        print("Multiplied: {}".format(AESs[(k,"neg")]*average_rates[(k,"
            ↪ neg")]))
        print()
```

A.5 Calculating Arrival Correlations

Listing A.5: Finding Arrival Correlations

```
def updates_during_period(updates, update_times, start, end):
    left_index = bisect.bisect_left(update_times, start)
    right_index = bisect.bisect_right(update_times, end)
    return updates[left_index: right_index + 1]

def updates_during_random_period(updates, update_times, starting_time,
    ↪ ending_time, size):
    total_length = (ending_time - starting_time).total_seconds()
    interval_start = starting_time + timedelta(seconds=np.random.uniform
        ↪ ()*(total_length - size))
    return updates_during_period(updates, update_times, interval_start,
        ↪ interval_start + timedelta(seconds=size))

def updates_during_random_period_multiple_dates(size):
    probabilities = [(e-s).total_seconds() - size for (s,e) in zip(
        ↪ starting_times, ending_times)]
    probabilities = [p/sum(probabilities) for p in probabilities]
    I = np.random.choice(len(all_updates), p=probabilities)
    return updates_during_random_period(all_updates[I], all_update_times
        ↪ [I], starting_times[I], ending_times[I], size)

K = 10
positive_updates = OrderedDict([(k,[]) for k in range(-K,K+1) if k !=
    ↪ 0])
negative_updates = OrderedDict([(k,[]) for k in range(-K,K+1) if k !=
    ↪ 0])
```

```
for _ in range(160000):
    updates = updates_during_random_period_multiple_dates(60)
    num_positive = OrderedDict([(k,0) for k in range(-K,K+1) if k != 0])
    num_negative = OrderedDict([(k,0) for k in range(-K,K+1) if k != 0])

    for u in updates:
        if u['change'] > 0:
            num_positive[u['k']] += 1
        else:
            num_negative[u['k']] += 1

    for k in range(-K,K+1):
        if k != 0:
            positive_updates[k].append(num_positive[k])
            negative_updates[k].append(num_negative[k])

observations = []
for k in positive_updates:
    observations.append(positive_updates[k])
for k in negative_updates:
    observations.append(negative_updates[k])
sigma = np.corrcoef(observations)
sigma
```

A.6 Generating Correlated Poisson Variables

Listing A.6: Generating Correlated Poisson Variables Using Copulas

```
def inverse_PDF_Poisson(average, u):
    x = 0
    p = np.exp(-average)
    s = p
    while u > s:
        x += 1
        p = p * average / x
        s = s + p
    return x

def mean_and_correlation(desired_correlation):
    n = 10000
    y = [np.random.multivariate_normal(mean=[0,0],cov=[[1,
       ↪ desired_correlation],[desired_correlation,1]]) \
         for _ in range(n)]
    y1,y2 = zip(*y)
    x1 = [inverse_PDF_Poisson(1,norm.cdf(y)) for y in y1]
    x2 = [inverse_PDF_Poisson(1,norm.cdf(y)) for y in y2]

    actual_correlation = np.corrcoef(x1,x2)[0][1]

    return np.mean(x1), np.mean(x2), actual_correlation
```

```
for corr in range(-10,11):
    c = corr / 10
    x1_mean, x2_mean, correlation = mean_and_correlation(c)
    print("Desired correlation: {}".format(c))
    print("x1 mean: {}".format(x1_mean))
    print("x2 mean: {}".format(x2_mean))
    print("Actual correlation: {}".format(correlation))
    print()
```

A.7 Simulation with Market Orders

Listing A.7: Simulating LOB Events And Market Orders

```
def arrivals_during_time_period(AESs, average_rates, correlations,
    ↪ start_time, T):
    correlated_normals = np.random.multivariate_normal(mean=[0 for _ in
        ↪ AESs], cov=correlations)
    us = [norm.cdf(y) for y in correlated_normals]
    num_arrivals = [inverse_PDF_Poisson(rate*T,u) for (rate,u) in zip(
        ↪ average_rates,us)]
    labels = [(k,"pos") for k in range(-K,K+1) if k != 0] + [(k,"neg")
        ↪ for k in range(-K,K+1) if k != 0]
    events = []
    for i,n in enumerate(num_arrivals):
        position, direction = labels[i]
        for _ in range(n):
            arrival_time = start_time + np.random.uniform()*T
            arrival_size = min(np.random.exponential(AESs[i]),5000)
            if direction == "neg":
                arrival_size *= -1
            events.append((arrival_time, arrival_size, position, False))
    return events

def simulate(starting_ref_price, starting_book, starting_time,
    ↪ num_intervals, T, A, market_orders, rates=None):
    ref_price = starting_ref_price
    sizes = copy.copy(starting_book)
    interval_start = starting_time

    # Find the market orders in each interval
    orders_in_interval = OrderedDict([(i,[]) for i in range(
        ↪ num_intervals)])
    i = 0
    for order in market_orders:
        if order[0] > starting_time + (i+1)*T:
            i += 1
        orders_in_interval[i].append(order)

    # Amount left to buy
    a = A
```

```
# Prices and amounts
filled_orders = []


for n in range(num_intervals):
    interval_start = n*T
    if rates == None:
        events = arrivals_during_time_period(AESs_vector,
            ↪ average_rates_vector, sigma, interval_start, T)
    else:
        events = arrivals_during_time_period(AESs_vector, rates[n],
            ↪ sigma, interval_start, T)
    orders = [(t,-size,1,True) for (t,size) in orders_in_interval[n
        ↪ ]]
    # Finish ordering all that is left
    if n == (num_intervals-1) and a > 0:
        orders.append(((n+1)*T,-a,1,True))
    all_events = sorted(events + orders, key=lambda x: x[0])

    for (t,change,k,is_order) in all_events:

        if is_order:
            left_to_fill = min(-change,a)
            p = round(ref_price + 0.5)
            while p < 617 and sizes[p] < left_to_fill:
                if sizes[p] > 0:
                    filled_orders.append((t,p,sizes[p]))
                left_to_fill -= sizes[p]
                sizes[p] = 0
                p += 1
            if left_to_fill > 0:
                filled_orders.append((t,p,left_to_fill))
            sizes[p] = max(0,sizes[p] - left_to_fill)
            left_to_fill = 0

            a = max(0, a + change)

        else:
            # Negative event
            if change < 0:
                # Check if first:
                if k < 0:
                    price = round(ref_price + 0.5 + k)
                    is_first = True
                    for p in reversed(range(price + 1,round(
                        ↪ ref_price + 0.5))):
                        if sizes[p] > 0:
                            is_first = False

                    # If first, go down prices
                    if is_first:
                        p = price
                        amount_left = -change
                        while i > 0 and sizes[p] < amount_left:
```

```
                        amount_left -= sizes[p]
                        sizes[p] = 0
                        p -= 1
                    sizes[p] = max(0, sizes[p] - amount_left)
                # Otherwise, just subtract
                else:
                    sizes[price] = max(0, sizes[price] + change)

            else:
                price = round(ref_price - 0.5 + k)
                is_first = True
                for p in range(round(ref_price + 0.5),price):
                    if sizes[p] > 0:
                        is_first = False

                # If first, go down prices
                if is_first:
                    p = price
                    amount_left = -change
                    while p < 1017 and sizes[p] < amount_left:
                        amount_left -= sizes[p]
                        sizes[p] = 0
                        p += 1
                    sizes[p] = max(0, sizes[p] - amount_left)
                else:
                    sizes[price] = max(0, sizes[price] + change)

        # Positive events
        else:
            if k < 0:
                price = round(ref_price + 0.5 + k)
            else:
                price = round(ref_price - 0.5 + k)
            sizes[price] += change

    # Update ref price
    for p in range(round(ref_price + 0.5),1017):
        if sizes[p] != 0:
            first_ask_price = p
            break
    for p in reversed(range(0,round(ref_price + 0.5))):
        if sizes[p] != 0:
            first_bid_price = p
            break

    if (first_ask_price - first_bid_price) % 2 != 0:
        ref_price = round(float(first_ask_price +
            ↪ first_bid_price) / 2,1)
    else:
        middle_price = round((first_ask_price + first_bid_price)
            ↪ /2)
        if middle_price < ref_price:
            ref_price = round(middle_price + 0.5, 1)
        else:
```

```
                    ref_price = round(middle_price - 0.5, 1)

    return filled_orders
```

A.8 Trade Execution Testing

A.8.1 TWAP Testing

Listing A.8: Performing TWAP Strategy Simulation

```
def simulate_split_orders(num_minutes, A, num_trades):
    order_size = A / num_trades
    market_orders = []
    for i in range(1,num_trades+1):
        market_orders.append(((num_minutes*60)*i/(num_trades+1),
            ↪ order_size))
    orders = simulate(average_price, starting_book, 0, num_minutes, 60,
        ↪ A, market_orders)
    return orders

orders = simulate_split_orders(10, 40000, 8)
print("VWAP:␣{}".format(vwap(orders)))
for t,price,volume in orders:
    print("{}␣{}␣{}".format(t,price,volume))
```

A.8.2 VWAP Testing

Listing A.9: Performing VWAP Strategy Simulation

```
def simulate_split_2rates_orders(num_minutes, A, num_trades,
    ↪ first_half_market_activity, first_half_trade_activity):
    order_size_average = A / num_trades
    order_size_first_half = order_size_average *
        ↪ first_half_trade_activity / 0.5
    order_size_second_half = order_size_average * (1 -
        ↪ first_half_trade_activity) / 0.5

    market_orders = []
    for i in range(1,num_trades+1):
        if i <= num_trades / 2:
            market_orders.append(((num_minutes*60)*i/(num_trades+1),
                ↪ order_size_first_half))
        else:
            market_orders.append(((num_minutes*60)*i/(num_trades+1),
                ↪ order_size_second_half))

    first_half_rates = [r*first_half_market_activity/0.5 for r in
        ↪ average_rates_vector]
```

```
    second_half_rates = [r*(1-first_half_market_activity)/0.5 for r in
        ↪ average_rates_vector]

    rates = OrderedDict([(n,None) for n in range(num_minutes)])
    for n in range(num_minutes):
        if n < num_minutes / 2:
            rates[n] = first_half_rates
        else:
            rates[n] = second_half_rates

    orders = simulate(average_price, starting_book, 0, num_minutes, 60,
        ↪ A, market_orders,rates=rates)
    return orders

orders = simulate_split_2rates_orders(10, 40000, 8, 0.75, 0.75)
print("VWAP: {}".format(vwap(orders)))
for t,price,volume in orders:
    print("{} {} {}".format(t,price,volume))
```

Bibliography

Alfonsi, A. and Schied, A. (2007). Optimal trade execution and absence of price manipulations in limit order book models. *Quantitative Finance*, 1(1):143–157.

Almgren, R. and Chriss, N. (1999). Optimal execution of portfolio transactions. *Journal of Risk*, 3(2):5–39.

Bae, T. and Kreinin, A. (2017). A backward construction and simulation of correlated poisson processes. *Journal of Statistical Computation and Simulation*, 87(8)(8):1593–1607.

Bouchaud, J.-P., Mezard, M., and Potters, M. (2002). Statistical properties of stock order books: Empirical results and models. *Quantitative Finance*, 2(4):251–256.

Coinbase (2018). Coinbase markets trading rules. `https://www.coinbase.com/legal/trading_rules?locale=en-US`. Accessed: December 28, 2018.

CoinMarketCap (2018). Coinbase pro market cap. `https://coinmarketcap.com/exchanges/coinbase-pro/`. Accessed: December 28, 2018.

Cont, R., Stoikov, S., and Talreja, R. (2010). A stochastic model for order book dynamics. *Operations Research*, 58(3):iii–772.

El Euch, O., Fukasawa, M., and Rosenbaum, M. (2018). The microstructural foundations of leverage effect and rough volatility. *Finance and Stochastics*, 22(2):241–280.

Forsyth, P., Kennedy, J., Tse, S., and Windcliff, H. (2012). Optimal trade execution: A mean quadratic variation approach. *Journal of Economic Dynamics and Control*, 36(12):1971–1991.

Gatheral, J. and Schied, A. (2011). Optimal trade execution under geometric brownian motion in the almgren and chriss framework. *International Journal of Theoretical and Applied Finance*, 14(3):353–368.

Hollifield, B., Miller, R. A., and Sands, P. (2004). Empirical analysis of limit order markets. *The Review of Economic Studies*, 71(4)(4):1027–1063.

Huang, W., Lehalle, C.-A., and Rosenbaum, M. (2013). Simulating and analyzing order book data: The queue-reactive model. *Journal of the American Statistical Association*, 110(509):107–122.

Inouye, D., Yang, E., Allen, G., and Ravikumar, P. (2017). A review of multivariate distributions for count data derived from the poisson distribution. *Wiley Interdisciplinary Reviews: Computational statistics*, 9(3):1–40.

Labadie, M. and Lehalle, C.-A. (2010). Optimal Algorithmic Trading and Market Microstructure. Technical report, CAMS.

Lim, M. and Coggins, R. (2005). Optimal trade execution: An evolutionary approach. *The 2005 IEEE Congress on Evolutionary Computation*, 2:1045–1052.

Nevmyvaka, Y., Feng, Y., and Kearns, M. (2006). Reinforcement learning for optimized trade execution. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 673–680, New York, NY, USA. ACM.

Obizhaeva, A. and Wang, J. (2013). Optimal trading strategy and supply/demand dynamics. *Journal of Financial Markets*, 16(1):1–32.

Python Software Foundation (2018). Copra. `https://pypi.org/project/copra/`. Accessed: December 28, 2018.

Ross, S. M. (2012). *Simulation, Fifth Edition*. Academic Press, Inc., Orlando, FL, USA.

Smith, E., Farmer, J. D., Gillemot, L., and Krishnamurthy, S. (2003). Statistical theory of the continuous double auction. *Quantitative Finance*, 3(6):481–514.

Spooner, T., Fearnley, J., Savani, R., and Koukorinis, A. (2018). Market making via reinforcement learning. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '18, pages 434–442, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.