Using Simulated Limit Order Book Data To Evaluate Optimal Trade Execution Algorithms: Interim Report

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

June 2019

Abstract

The electronic nature of markets today has allowed traders to infer information from limit order book (LOB) data and execute trades based on this information. In this paper, I attempt to solve the problem of optimal trade execution using simulated market data. Given the current state of the limit order book for a particular asset and the amount to buy (or sell), I model the problem as a Markov decision process and output the orders to place at each time step. The simulated environment allows me to evaluate a trading strategy while updating LOB dynamics in real time to take into account the agent's market impact. I first use high frequency LOB data to simulate LOB dynamics with real-time market impact adjustments using a modified version of the queue reactive model developed by Huang et al. (2013). I then evaluate the performance of common trade execution strategies in the simulated environment. The findings from this paper are relevant for institutions who want to execute trades while minimizing cost as well as those interested in simulating trades in a market.

Acknowledgements

I will write this when I finish writing the thesis.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Summary of Progress

In this thesis, I am building a market simulator that is based on a LOB queuing model to test various optimal trade execution strategies. This process involves developing a model that accurately reflects real world dynamics of the LOB, estimating the parameters, and writing a program for simulation that can take decisions by an agent and update in real time. I am using data from trades on Ethereum Classic cryptocurrency from the Coinbase Pro exchange. My progress thus far can be summarized as follows:

- **Software**: I have organized my code in a github repository
- **Refined Data Collection**: I have written code to collect and store updates. I have collected several days worth of data in the months of December and January.
- **Data Processing**: I have written code to maintain an internal LOB for analysis
- **Data Analysis and Model Development**: I started with a queuing model developed by Huang et al. (2013) and modified it based on what I have analyzed in the data. This includes:
  - Verifying that inter-arrival times are approximately exponential and estimating the average rate of arrival

  - Estimating average event sizes and allowing for variable size events
  - Finding correlations between the Poisson arrivals

- **Simulation Algorithm**: I have developed the simulation algorithm and begun writing preliminary code for parts of it (such as generating correlated Poisson arrivals using copulas and backward simulation)

- **Thesis Writing**: I have written rough drafts of the introduction, literature review, data description, and experimental procedure for the data analysis described above. I also have the boilerplate for the final product written in LaTeX.

Chapter 2

Timeline

2.1 Summary of Major Work that Needs to be Done

- Continue collecting data
- Estimate parameters with full dataset
- Write code to simulate LOB with estimated parameters and agent decisions
- Run trading strategies in simulated environment and evaluate performance,
- Write results in thesis

2.2 Rough Timeline

- By the end of February, I will have the full data-set that spans several weeks (I already have code to estimate the parameters).
- By the end of February, I also hope to have market simulation code written as described by the algorithm in 6.

- By mid-March, I hope to have analyzed the performance of various trade execution strategies (completing the experimental procedure)

- I will spend the rest of March and April writing the thesis.

Chapter 3

Introduction

3.1 Problem Description

This thesis analyzes a problem in quantitative finance known as the optimal trade execution problem. In essence, when trading a large volume of a security in a short amount of time, I seek to minimize slippage, which is the difference between the expected price of a security and the price at which the trade is actually executed. More formally, given a volume V of a security to buy and a time limit T, I seek to limit the amount of money spent to acquire these shares. Similarly, we could be given an amount V to sell and a time limit T, and seek to maximize the revenue received from selling the security.

This problem is most relevant to a large institutional investor, such as a hedge fund, who may want to buy (respectively, sell) a large volume of a security because they believe that the price of the security will rise (fall), but find that there is not enough supply (demand) at the desired price level to satisfy the full order. If the full trade could be executed at the desired price, the hedge fund would want to execute the trade immediately in order to capture as much of the expected change in price as possible. However, the hedge fund would have to execute part of the trade at more unfavorable prices. They can instead choose to split the order over a longer time period to minimize slippage. In general, there is a trade-off between how quickly the desired position is achieved and the price impact of the trade.

3.2 Background

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). For example, part of the LOB for Coca-Cola (KO) on the NYSE may look like Table 3.1, in which the 5 best bids and asks are listed:

Table 3.1: 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 could be considered \$48.695, which is the midpoint between the best bid and best ask price.

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 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. It could also place a limit 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 market order from the ask side. The updated LOB is shown in Table 3.2.

Table 3.2: Coca-Cola Updated Limit Order Book

Bids		**Asks**	
Volume	Price	Price	Volume
1000	48.69	48.70	500
2000	48.68	48.71	1500
$\underline{10000}$	$\underline{48.67}$	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 unlike the market order.

Of course, it could also place multiple market or limit orders over time to achieve the desired position while minimizing market impact. Given a time horizon T and the state of the order book at each time step, we seek to find the optimal placement of market and limit orders to achieve the desired position. I will use the common industry benchmark of Volume Weighted Average Price (VWAP) to measure performance of our trading strategy:

$$\frac{\sum \text{Volume} * \text{Price}}{\sum \text{Volume}}$$

In the case of market order example, the VWAP would be (500*48.70 + 1500*48.71 + 3500*48.72 + 1500*48.75)/7000 = \$48.723/share.

Chapter 4

Preliminary Literature Review

4.1 Past Approaches of Solving Optimal Trade Execution

Several approaches have been taken in the past to tackle the optimal trading execution problem. Almgren and Chriss (1999) developed a theoretical model for optimal trade execution based on maximizing expected utility at different levels of risk aversion. Laruelle et al. (2013) used a stochastic recursive procedure to calculate the optimal distance from the bid-ask spread to post passive limit orders. Studies have also been conducted using machine learning where models are back tested on historical data. Lim and Coggins (2005) used genetic algorithms to develop an optimal trading strategy using Australian Stock Exchange data. Nevmyvaka et al. (2006) used reinforcement learning to solve the problem for several stocks listed on the NASDAQ. Its methodology is summarized below:

Once again, the goal of the agent (the institution that is attempting to optimally trade) was to buy V shares in a time horizon T. First, the time horizon was divided into discrete time steps, where any number of shares could be bought at each step. Reinforcement learning is a state-based strategy, meaning that the agent examines the state of the Markovian system to decide what to do next. The state of the system is modelled by two private variables that represent the number of shares left to buy and the amount of time left, as well as several market variables that represent various properties of the LOB. At each time step, an action could be taken, which was to place a market or limit order at a specified volume. At the end, all remaining shares had to be bought/sold in a market order. After each time step, a reward is then received, which is the cash inflow if selling or negative outflow if buying. In each state, every possible action is attempted and one is chosen that maximizes expected reward. The system then transitions to the next state and updates the expected reward function. The study based awards and market variables on historical high-frequency NASDAQ data, and assumed that the actions of the agent did not affect future dynamics of the order book.

This thesis seeks to evaluate trading strategies that attempt to solve optimal trade execution problem, but with the impact of the agents actions taken into account. In order to accomplish this, we must first simulate the LOB while updating its dynamics when a trade occurs. To do so, we simulate the LOB using a modified version of the queue reactive model of the LOB by Huang et al. (2013), which is described in Section 4.2.

4.2 Queue Reactive Model

In this section, we describe the queue reactive model used to simulate the LOB that is based on the one developed by Huang et al. (2013). The LOB is modelled as a $2K$-dimensional vector Q, where K is number of limits on each side of the Markovian queuing system. Let p_{ref} be the reference price of the of the stock. Then, $[Q_k : k = -1 \dots -K]$ represents the number of orders on the bid side $k - 0.5$ ticks to the left of p_{ref} and $[Q_k : k = 1 \dots K]$ represents the number of orders on the ask side $k - 0.5$ ticks to the right of p_{ref}. Then, $X(t) = (q_{-K}(t), q_{-1}(t), q_1(t), , q_K(t))$ is a continuous-time Markov jump process with a jump size equal to one.

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

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

We choose p_{ref} to be a price in between 2 adjacent ticks, so that an events at each position occur at actual ticks. Let b be the best bid price and a be the best ask price. If b and a are an odd number of ticks apart, then we have

$$p_{ref} = (b + a)/2$$

If b and a are an even number of ticks apart, then we could pick p_{ref} to be $(b+a)/2+0.5$ or $(b + a)/2 - 0.5$. We choose the one that is closest to the previous reference price.

If an event occurs that changes the best bid or ask, then we update p_{ref}. It can be seen that under certain conditions, this model is an ergodic Markov process (see Huang et al. (2013)). We can then use historical LOB data to estimate AES_k, $\lambda_k^+(q)$, and $\lambda_k^-(q)$ for each k.

4.3 Market Simulation

Studies in the past have also used models similar to the queue reactive model by modelling arrivals as Poisson processes. For example, El Euch et al. (2018) modelled price jumps of an asset using a bi-dimensional Hawkes process. Simulation also includes correlated Poisson processes, whose construction uses copulas as described by Inouye et al. (2017) and backwards simulation as described by Bae and Kreinin (2017).

Chapter 5

Data Source

5.1 Coinbase Exchange

To simulate a real world security, I use high frequency LOB data of cryptocurrency securities from Coinbase Pro. Coinbase Pro (formerly known as Global Digital Asset Exchange or GDAX) is the advanced trading platform of the 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, Litecoin, while fiat-currencies are typically the USD or Euro. As of November 2018, Coinbase Pro has a total daily trading volume of about \$150 million (CoinMarketCap (2018)). I choose to use Coinbase Pro 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 that I model in this paper may not entirely reflect those of securities from other exchanges such as stocks, futures, etc., I hope that the methodology in this paper is easily adaptable given LOB data of any type of security. The trading rules of Coinbase Pro are representative of most exchanges. Coinbase Pro allows limit and market orders with time priority. It also allows stop orders, which is an order to place a limit or market order when the price of the security reaches a specified price. Coinbase Pro also 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 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 (Coinbase (2018)).

5.2 Data Collection

Data is collected through the Coinbase Pro API using a Python library called CoPrA, which is an asynchronous web socket client (Foundation (2018)). Specifically, I use Coinbase Pro's "level2" channel, which provides me with a snapshot of the LOB. It also provides updates at every position of the LOB whenever an order happens. See listing 8.2 the code written. Using these updates, I can maintain an internal LOB to use for data analysis. I have collected data for several days in the months of December and January of all the updates on the LOB of Ethereum Classic using USD (ETC-USD ticker), which has around $2 million volume of trades each day. For example, the data for December 30, 2018 is shown below, which contained about 330000 updates, where the initial LOB is shown in Figure 5.1. The 10 best bids and asks are shown where the vast majority of updates are made, but the LOB also features prices closer to the reference price.

This LOB is typical for ETC-USD, with most of the positions near the reference

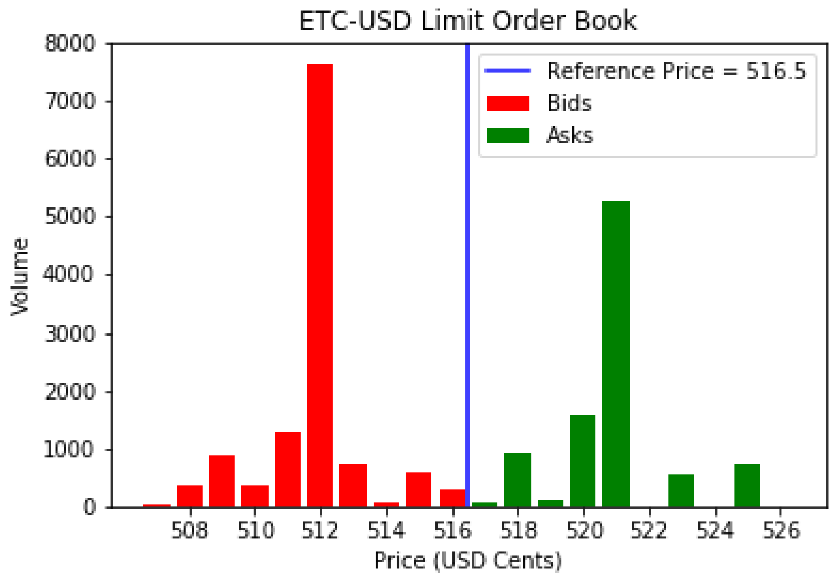
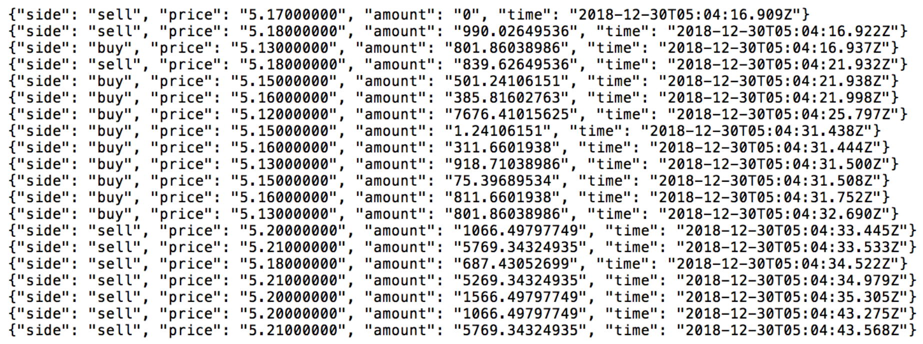


Figure 5.1: LOB Sample

price filled. The bid and ask sides are relatively balanced, meaning that the reference price is stable. This is not always the case, as the bid or ask side could have significantly more volume when the price shifts. The first 20 updates are also shown in Figure 5.2.

As can be seen, the majority of the updates are given to the nearest millisecond and the majority of the updates occur near the reference price and updates are given to the nearest millisecond.



```
{"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"}
```

Figure 5.2: LOB Updates

Chapter 6

Experimental Procedure

6.1 Simulating the LOB

6.1.1 Queuing Model

Here, I will describe the queuing model used in this thesis to simulate the LOB. It is based on the one developed by Huang et al. (2013) but with several important modifications. I again represent the LOB as a $2K$-dimensional vector $X(t) = (q_{-K}(t), q_{-1}(t), q_1(t), , q_K(t))$ with the center corresponding to the reference price. Events at each position k are either positive or negative with the magnitude distributed exponentially with means μ_k^+ and μ_k^- respectively. That is, if a positive event occurs at time t with size e, set $q(t) \leftarrow q(t) + e$, and if a negative event occurs at time t with size e, set $q(t) \leftarrow (q(t) - e)^+$ where $b^+ = 0$ if $b < 0$ and b otherwise (so as to keep the queue size non negative). An exception occurs when a negative order occurs at the best bid or ask. I assume it is a market order and the size is larger than the volume, fill the leftover with the next best prices. I estimate μ_k^+ and μ_k^- as the AES for positive and negative arrivals. Arrivals of events are jointly distributed as a multivariate Poisson process. There are $4K$ such processes, with positive and negative arrivals for $2K$ positions. The marginal arrival process is a Poisson process with positive and negative arrivals having mean rates λ_k^+ and λ_k^- respectively. I simulate arrivals in time periods of length b. Let $N_{k+}(b)$ be the number of positive arrivals at position k ($N_{k-}(b)$ for negative arrivals) during a time period of length b. Then, the arrivals have a $4K \times 4K$ correlation matrix σ where σ_{i+j+} is the $corr(N_{i+}(b), N_{j+}(b))$ (same for negative arrivals).

Here, I will highlight the differences between this model and the queue-reactive model developed by Huang et al. (2013) and explain the reasons for the modifications. First, the size of positive and negative events are not just the AES, but rather is a continuous random variable. This is because the event sizes that I found have considerable variation. I find in subsection 6.1.3 that an exponential distribution reasonably fits the distribution of event sizes. Although it may not fit the data completely, this part of the model is the easiest to adjust.

Second, I do not vary the rate of arrival as the size of the queue changes. This is because I did not find a substantial change in rate as the queue size changed, and the queue size varied widely (sometimes several dozens multiplies of the AES) and I did not have enough data points to estimate the rate at some sizes. It also simplifies the model greatly.

Third, I model the arrival processes as a multivariate Poisson process instead of independent Poisson processes. In subsection 6.1.4, I verify the claim that the arrival rates at each position reasonably follow a Poisson process. However, I find in subsection 6.1.5 that these processes are significantly correlated.

6.1.2 Building An Abbreviated Order Book

Although I maintain the full order book after each update, the queuing model only contains the first K prices on the bid and ask side closest to the reference price. I therefore only estimate parameters for these positions. Usually, the reference price is the midpoint between the best bid and best ask price. However, if the best bid and ask are an even distance apart, I add or subtract 0.5 to make it closest to the previous reference price. I then restart the process of recording data.

Often in the data set, several orders of the same magnitude, sign, and inter-arrival time occur in quick succession. This is likely a trader who has split up a larger trade for whatever reason. As it should represent one trade, I combine orders that occur in quick succession (defined as occurring less than 0.01 seconds after the last) if they are at the same position and sign. See listing 8.3 for the code written to process the data.

6.1.3 Average Event Size Estimate

In this section, I first examine whether the event sizes can reasonably fitted with exponential distribution with rate equal to the AES.

As can be seen from Figure 6.1, the exponential distribution with the mean rate as the AES fits the data relatively well. However, there are some discrepancies such as the large frequency of orders at 500. A possible explanation for this is a trader splitting up a trade into equally sized amounts. There are also a non-negligible of orders near 5000 since that is the upper limit order size imposed by Coinbase. I choose to ignore these discrepancies so that the model can be more broadly applicable, but it may be possible to more accurately model the dynamics of the ETC-USD Coinbase Pro LOB with these taken into account. The AES is reported for $K = 10$ in table 6.1. It can be seen that in general, as the price gets closer to the reference price, the AES decreases, but the number of events increases.

6.1.4 Average Arrival Rates

We first examine whether event arrivals at each position can be modelled as a Poisson process. To do so, I test whether the inter-arrival times of the events are exponentially distributed. It can be seen from the QQ-plots in Figures 6.2 and 6.3, which compare

Figure 6.1: Event Sizes Compared to Exponential Distribution (4 Positions Closest to Reference Price)

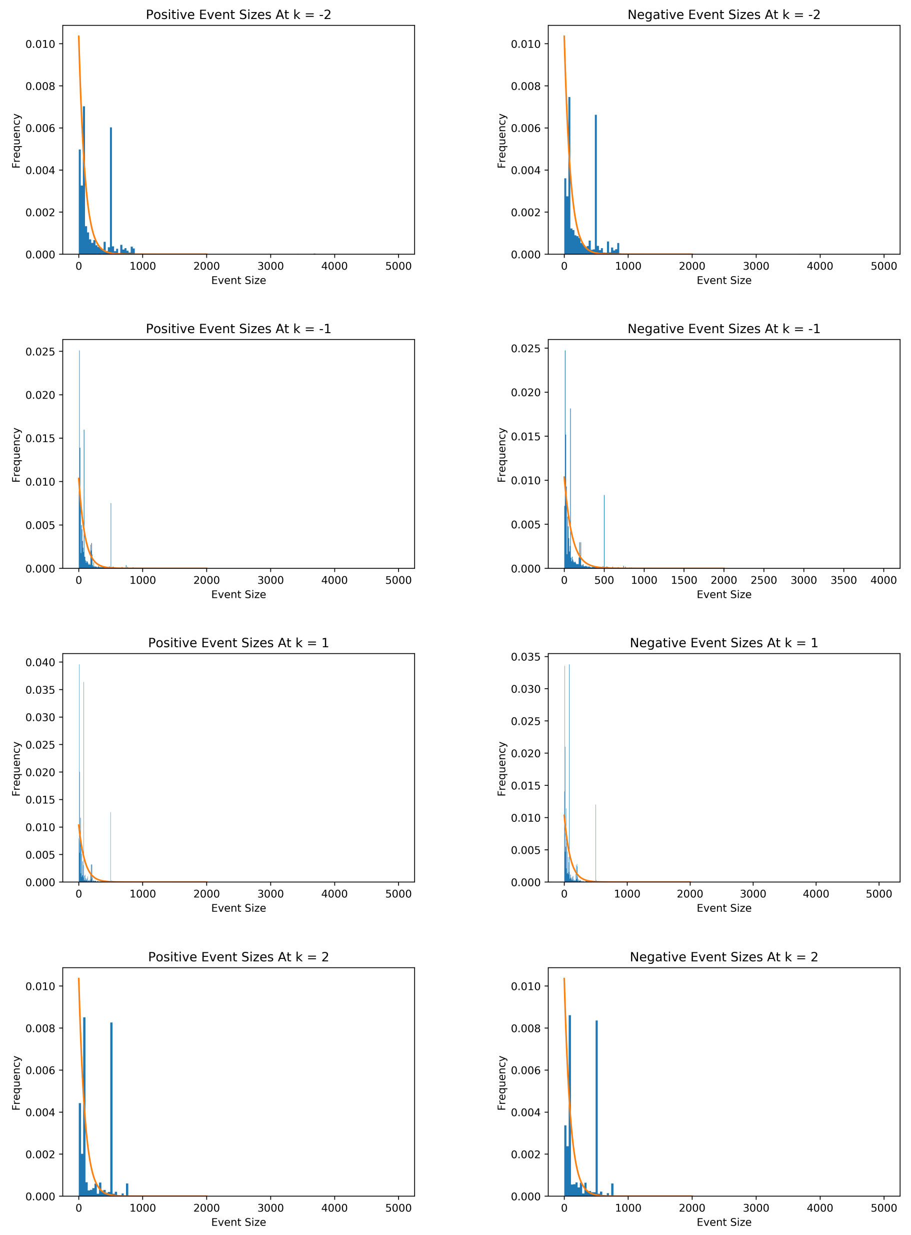


Table 6.1: Average Event Sizes using Data from December 30, 2018

	Positive Events		**Negative Events**	
k	n	*AES*	n	*AES*
-10	138	536.72	205	550.86
-9	276	331.13	302	280.93
-8	388	534.59	430	659.51
-7	1249	566.10	1207	577.19
-6	4145	520.49	4017	537.98
-5	7615	471.58	7655	470.21
-4	8958	463.97	9115	453.16
-3	11584	405.31	11837	399.33
-2	19494	247.61	18324	269.66
-1	20666	117.95	18028	121.73
1	19064	96.56	18402	94.21
2	13420	248.50	13641	255.25
3	13214	356.71	13643	350.755
4	15322	327.24	15119	335.16
5	12444	325.70	12090	324.47
6	4733	460.40	4588	466.64
7	1466	620.93	1400	663.58
8	428	562.84	410	545.73
9	229	502.40	218	577.02
10	168	340.23	174	287.10

the empirical distribution to an exponential distribution, that the inter-arrival times have heavy right tails compared to the exponential distribution. When I plot the histogram of inter-arrival times vs. the an exponential distribution, it can be seen that the exponential distribution fits the data well for the majority of the points except for a small number of outliers to the right. Despite this discrepancy, I choose to model the arrival as a Poisson process because the number of outliers is small and for its desirable properties for ease of simulation. I report the average rates λ_k^+ and λ_k^- for $K = 10$ in Table 6.2

Table 6.2: Average Arrival Rates (s^{-1}) using Data from December 30, 2018

k	λ_k^+	λ_k^-
-10	0.001597	0.002373
-9	0.003194	0.0034952
-8	0.004491	0.004977
-7	0.01446	0.01397
-6	0.04797	0.04649
-5	0.08813	0.08860
-4	0.1037	0.1055
-3	0.1341	0.1370
-2	0.2256	0.2121
-1	0.2392	0.2087
1	0.2206	0.2130
2	0.1553	0.1579
3	0.1529	0.1579
4	0.1773	0.1750
5	0.1440	0.1399
6	0.05478	0.05310
7	0.01697	0.01620
8	0.004954	0.004745
9	0.002650	0.002523
10	0.001944	0.002014

Figure 6.2: Inter-Arrival Times Compared to Exponential Distribution using Data from December 30, 2018

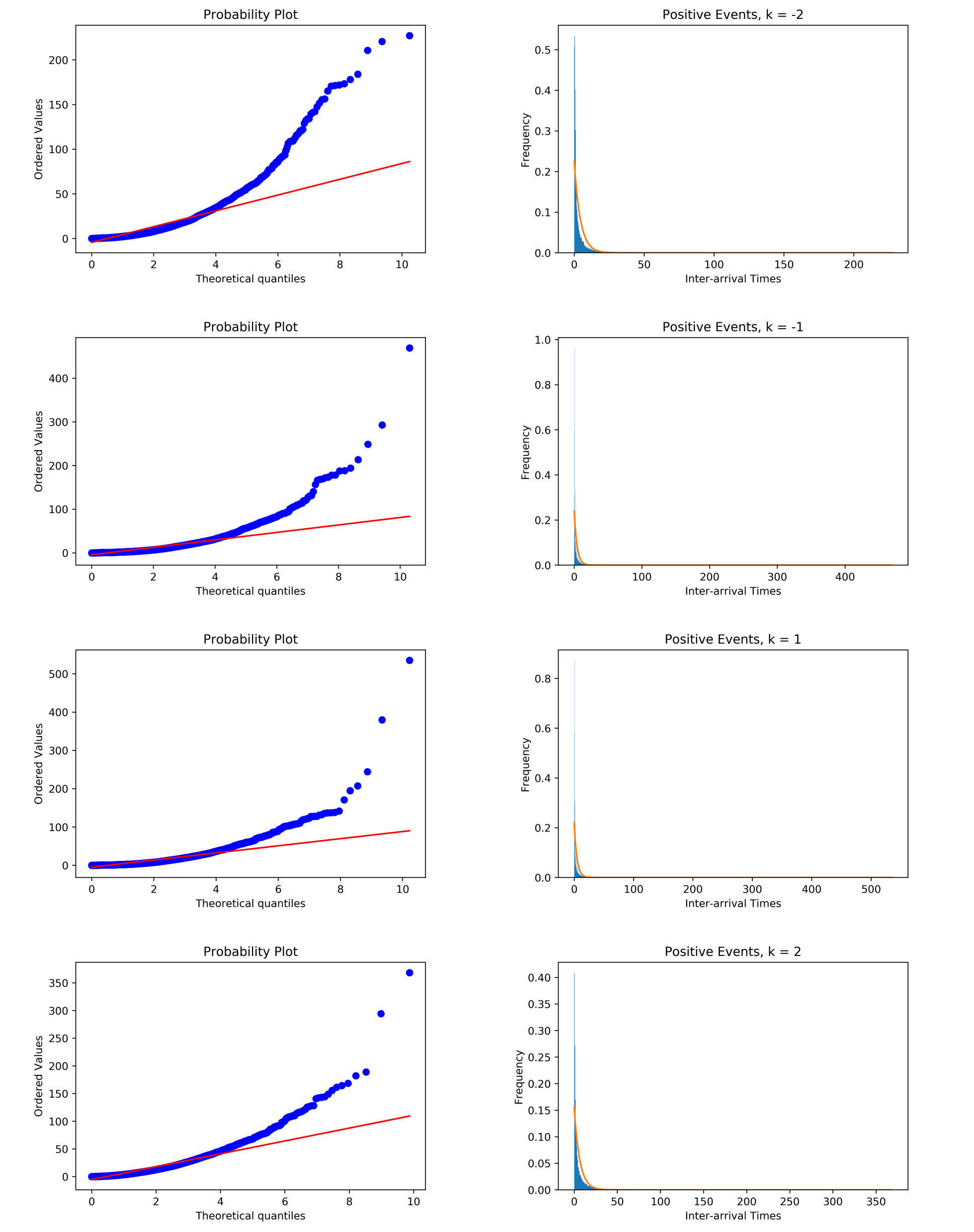
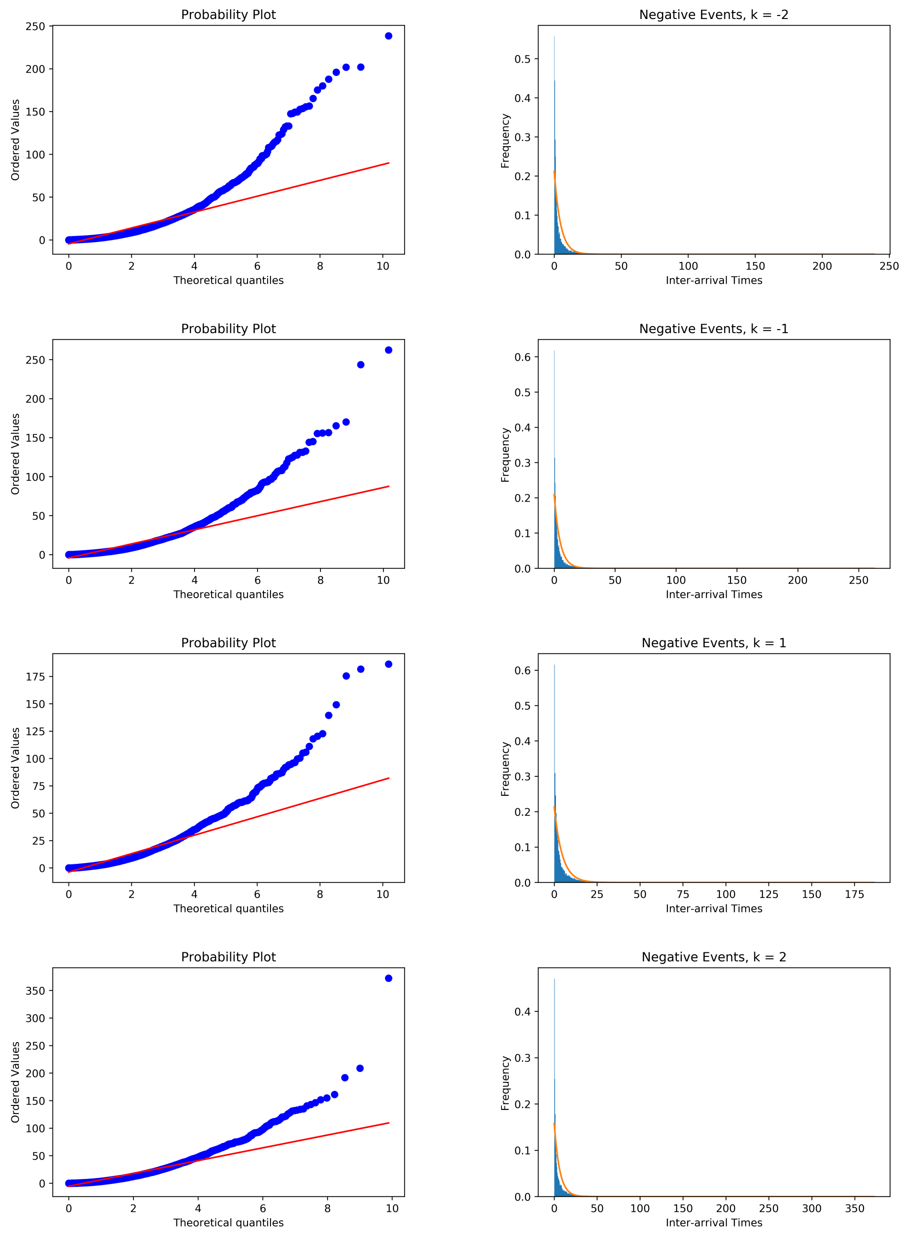
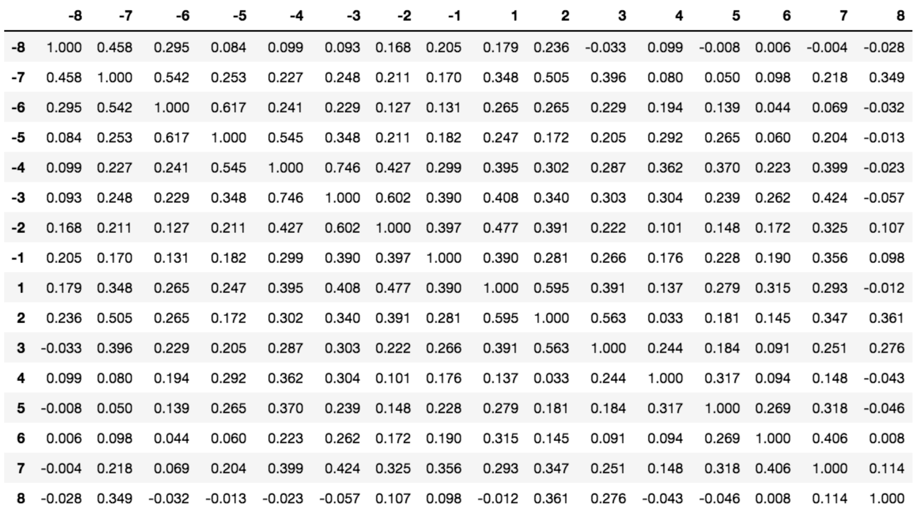


Figure 6.3: Inter-Arrival Times for Negative Events Compared to Exponential Distribution using Data from December 30, 2018





	-8	-7	-6	-5	-4	-3	-2	-1	1	2	3	4	5	6	7	8
-8	1.000	0.458	0.295	0.084	0.099	0.093	0.168	0.205	0.179	0.236	-0.033	0.099	-0.008	0.006	-0.004	-0.028
-7	0.458	1.000	0.542	0.253	0.227	0.248	0.211	0.170	0.348	0.505	0.396	0.080	0.050	0.098	0.218	0.349
-6	0.295	0.542	1.000	0.617	0.241	0.229	0.127	0.131	0.265	0.265	0.229	0.194	0.139	0.044	0.069	-0.032
-5	0.084	0.253	0.617	1.000	0.545	0.348	0.211	0.182	0.247	0.172	0.205	0.292	0.265	0.060	0.204	-0.013
-4	0.099	0.227	0.241	0.545	1.000	0.746	0.427	0.299	0.395	0.302	0.287	0.362	0.370	0.223	0.399	-0.023
-3	0.093	0.248	0.229	0.348	0.746	1.000	0.602	0.390	0.408	0.340	0.303	0.304	0.239	0.262	0.424	-0.057
-2	0.168	0.211	0.127	0.211	0.427	0.602	1.000	0.397	0.477	0.391	0.222	0.101	0.148	0.172	0.325	0.107
-1	0.205	0.170	0.131	0.182	0.299	0.390	0.397	1.000	0.390	0.281	0.266	0.176	0.228	0.190	0.356	0.098
1	0.179	0.348	0.265	0.247	0.395	0.408	0.477	0.390	1.000	0.595	0.391	0.137	0.279	0.315	0.293	-0.012
2	0.236	0.505	0.265	0.172	0.302	0.340	0.391	0.281	0.595	1.000	0.563	0.033	0.181	0.145	0.347	0.361
3	-0.033	0.396	0.229	0.205	0.287	0.303	0.222	0.266	0.391	0.563	1.000	0.244	0.184	0.091	0.251	0.276
4	0.099	0.080	0.194	0.292	0.362	0.304	0.101	0.176	0.137	0.033	0.244	1.000	0.317	0.094	0.148	-0.043
5	-0.008	0.050	0.139	0.265	0.370	0.239	0.148	0.228	0.279	0.181	0.184	0.317	1.000	0.269	0.318	-0.046
6	0.006	0.098	0.044	0.060	0.223	0.262	0.172	0.190	0.315	0.145	0.091	0.094	0.269	1.000	0.406	0.008
7	-0.004	0.218	0.069	0.204	0.399	0.424	0.325	0.356	0.293	0.347	0.251	0.148	0.318	0.406	1.000	0.114
8	-0.028	0.349	-0.032	-0.013	-0.023	-0.057	0.107	0.098	-0.012	0.361	0.276	-0.043	-0.046	0.008	0.114	1.000

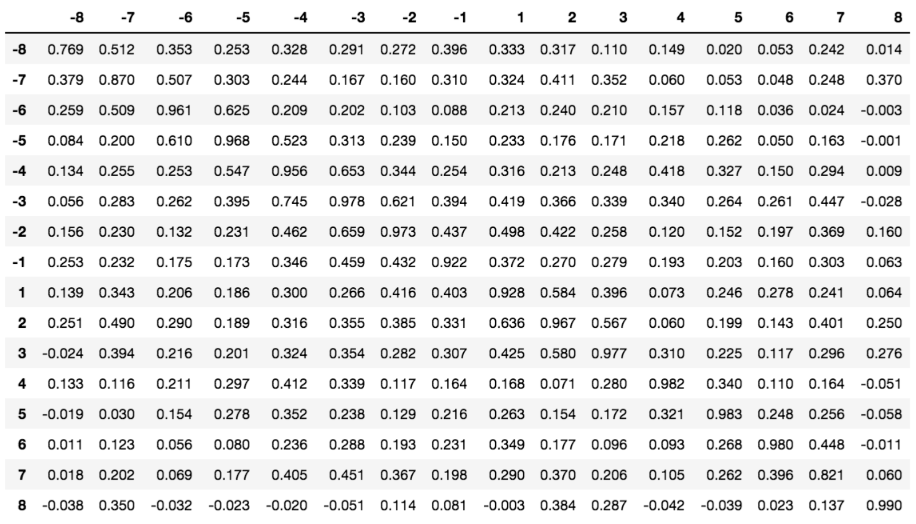
Figure 6.4: Correlation Matrix for $N_i^+(t), N_j^+(t)$, $t = 60$ seconds

6.1.5 Arrival Correlations

Although each of the marginal processes can be modelled as a Poisson process, I found that there were significant non-zero correlations between the arrivals at each position. Using a time period b of 60 seconds, I found the correlation between $N_i^{\pm}(b)$ and $N_j^{\pm}(b)$ by sampling random intervals of length t through the period of data collection. The correlation matrix for positive events (closest 8 positions) is shown in Figure 6.4 where the entry (i, j) is $corr(N_i^+(b), N_j^+(b))$. In general, the correlations are positive, with the magnitude of the correlation decreasing as distance from the positions increase.

The correlation matrix for positive and negative events is shown in Figure 6.5 where the entry (i, j) is $corr(N_i^+(b), N_j^-(b))$. Particularly notable is the very high (nearly 1) correlation between positive and negative at the same position. This 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 6.6 where the entry (i, j) is $corr(N_i^-(b), N_j^-(b))$. This matrix is similar in nature to the one for positive events.



	-8	-7	-6	-5	-4	-3	-2	-1	1	2	3	4	5	6	7	8
-8	0.769	0.512	0.353	0.253	0.328	0.291	0.272	0.396	0.333	0.317	0.110	0.149	0.020	0.053	0.242	0.014
-7	0.379	0.870	0.507	0.303	0.244	0.167	0.160	0.310	0.324	0.411	0.352	0.060	0.053	0.048	0.248	0.370
-6	0.259	0.509	0.961	0.625	0.209	0.202	0.103	0.088	0.213	0.240	0.210	0.157	0.118	0.036	0.024	-0.003
-5	0.084	0.200	0.610	0.968	0.523	0.313	0.239	0.150	0.233	0.176	0.171	0.218	0.262	0.050	0.163	-0.001
-4	0.134	0.255	0.253	0.547	0.956	0.653	0.344	0.254	0.316	0.213	0.248	0.418	0.327	0.150	0.294	0.009
-3	0.056	0.283	0.262	0.395	0.745	0.978	0.621	0.394	0.419	0.366	0.339	0.340	0.264	0.261	0.447	-0.028
-2	0.156	0.230	0.132	0.231	0.462	0.659	0.973	0.437	0.498	0.422	0.258	0.120	0.152	0.197	0.369	0.160
-1	0.253	0.232	0.175	0.173	0.346	0.459	0.432	0.922	0.372	0.270	0.279	0.193	0.203	0.160	0.303	0.063
1	0.139	0.343	0.206	0.186	0.300	0.266	0.416	0.403	0.928	0.584	0.396	0.073	0.246	0.278	0.241	0.064
2	0.251	0.490	0.290	0.189	0.316	0.355	0.385	0.331	0.636	0.967	0.567	0.060	0.199	0.143	0.401	0.250
3	-0.024	0.394	0.216	0.201	0.324	0.354	0.282	0.307	0.425	0.580	0.977	0.310	0.225	0.117	0.296	0.276
4	0.133	0.116	0.211	0.297	0.412	0.339	0.117	0.164	0.168	0.071	0.280	0.982	0.340	0.110	0.164	-0.051
5	-0.019	0.030	0.154	0.278	0.352	0.238	0.129	0.216	0.263	0.154	0.172	0.321	0.983	0.248	0.256	-0.058
6	0.011	0.123	0.056	0.080	0.236	0.288	0.193	0.231	0.349	0.177	0.096	0.093	0.268	0.980	0.448	-0.011
7	0.018	0.202	0.069	0.177	0.405	0.451	0.367	0.198	0.290	0.370	0.206	0.105	0.262	0.396	0.821	0.060
8	-0.038	0.350	-0.032	-0.023	-0.020	-0.051	0.114	0.081	-0.003	0.384	0.287	-0.042	-0.039	0.023	0.137	0.990

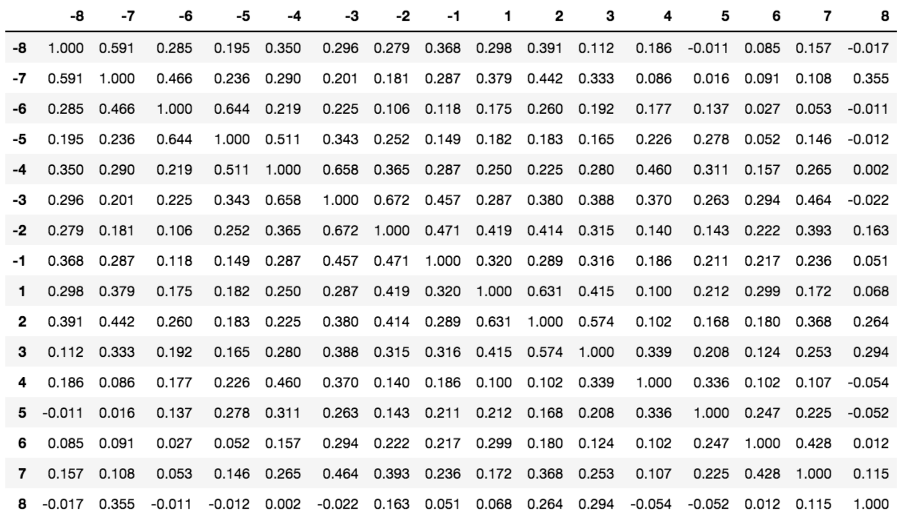
Figure 6.5: Correlation Matrix for $N_i^+(b), N_j^-(b)$, $b = 60$ seconds

These non-zero correlations are clearly non-negligible, so I model the arrivals are a multivariate Poisson process where the arrivals are correlated and each marginal arrival distribution is Poisson. The correlations can be combined to form σ as described in subsection 6.1.1. I will discuss the procedure for simulating the multivariate Poisson process in Subsection 6.1.6.

6.1.6 Simulation Procedure

In presenting the simulation procedure, it is necessary to discuss the method for generating correlated Poisson processes. I use a method similar to the ones discussed by Inouye et al. (2017) to generate multivariate Poisson variables with desired desired correlations.

The goal is to generate Poisson random variables with means $x_1, x_2, \ldots, x_d$ with means $\lambda_1, \lambda_2, \ldots, \lambda_d$ with correlations defined by the matrix σ. To do so, I make use of a copula. A copula is defined as any joint probability distribution function where



	-8	-7	-6	-5	-4	-3	-2	-1	1	2	3	4	5	6	7	8
-8	1.000	0.591	0.285	0.195	0.350	0.296	0.279	0.368	0.298	0.391	0.112	0.186	-0.011	0.085	0.157	-0.017
-7	0.591	1.000	0.466	0.236	0.290	0.201	0.181	0.287	0.379	0.442	0.333	0.086	0.016	0.091	0.108	0.355
-6	0.285	0.466	1.000	0.644	0.219	0.225	0.106	0.118	0.175	0.260	0.192	0.177	0.137	0.027	0.053	-0.011
-5	0.195	0.236	0.644	1.000	0.511	0.343	0.252	0.149	0.182	0.183	0.165	0.226	0.278	0.052	0.146	-0.012
-4	0.350	0.290	0.219	0.511	1.000	0.658	0.365	0.287	0.250	0.225	0.280	0.460	0.311	0.157	0.265	0.002
-3	0.296	0.201	0.225	0.343	0.658	1.000	0.672	0.457	0.287	0.380	0.388	0.370	0.263	0.294	0.464	-0.022
-2	0.279	0.181	0.106	0.252	0.365	0.672	1.000	0.471	0.419	0.414	0.315	0.140	0.143	0.222	0.393	0.163
-1	0.368	0.287	0.118	0.149	0.287	0.457	0.471	1.000	0.320	0.289	0.316	0.186	0.211	0.217	0.236	0.051
1	0.298	0.379	0.175	0.182	0.250	0.287	0.419	0.320	1.000	0.631	0.415	0.100	0.212	0.299	0.172	0.068
2	0.391	0.442	0.260	0.183	0.225	0.380	0.414	0.289	0.631	1.000	0.574	0.102	0.168	0.180	0.368	0.264
3	0.112	0.333	0.192	0.165	0.280	0.388	0.315	0.316	0.415	0.574	1.000	0.339	0.208	0.124	0.253	0.294
4	0.186	0.086	0.177	0.226	0.460	0.370	0.140	0.186	0.100	0.102	0.339	1.000	0.336	0.102	0.107	-0.054
5	-0.011	0.016	0.137	0.278	0.311	0.263	0.143	0.211	0.212	0.168	0.208	0.336	1.000	0.247	0.225	-0.052
6	0.085	0.091	0.027	0.052	0.157	0.294	0.222	0.217	0.299	0.180	0.124	0.102	0.247	1.000	0.428	0.012
7	0.157	0.108	0.053	0.146	0.265	0.464	0.393	0.236	0.172	0.368	0.253	0.107	0.225	0.428	1.000	0.115
8	-0.017	0.355	-0.011	-0.012	0.002	-0.022	0.163	0.051	0.068	0.264	0.294	-0.054	-0.052	0.012	0.115	1.000

Figure 6.6: Correlation Matrix for $N_i^-(b), N_j^-(b)$, $b = 60$ seconds

each marginal distribution is normally distributed Ross (2012). That is,

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

is a copula if $C(0,0,\ldots,0) = 0$ and $C(1,1,u_i,1,\ldots,1) = u_i$ for all i. I will use the Gaussian copula as described in Inouye et al. (2017) where

$$C_\sigma^{Gaussian}(u_1,u_2,\ldots,u_d) = H_\sigma(H^{-1}(u_1), H^{-1}(u_2),\ldots,H^{-1}(u_d))$$

where H is the inverse cumulative distribution function for a standard random normal variable and H_σ is the cumulative distribution function for a random multivariate normal vector with mean 0 and correlation matrix R. I can generate u by first generating a multivariate normal random variable Y with correlation matrix σ and setting $(u_1,u_2,\ldots,u_d) = (H^{-1}(u_1), H^{-1}(u_2),\ldots,H^{-1}(u_d))$. I then pair it 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. I can then set

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

To implement $F^-1_i(u_i)$, I will use the inverse transform method described in Ross (2012):

Algorithm 1: Inverse Transform Method for Poisson Variable With Rate λ and input u

```
$i = 0$;
$p = e^{-\lambda}$;
$F = p$;
**while** $u >= F$ **do**
    $p = \lambda p/(i+1)$ ;
    $F = F + p$ ;
    $i = i + 1$ ;
**end**
return $i$ ;
```

Each x_i will have marginal have a marginal Poisson distribution with rate λ_i and x will approximately have correlation matrix σ. For code demonstrating the construction of correlated Poisson processes, see listing 8.4.

Now, I can simulate arrivals over a time period T. I will use backwards-simulation as described in Bae and Kreinin (2017) to simulate arrivals in successive smaller chunks of size t. Namely, I will generate x_i^+ and x_i^- with means $\lambda_i^+ * t$ and $\lambda_i^- * t$ respectively and correlation matrix σ. Then, I use the fact that conditioned on the number of arrivals in a time period t being equal to n, these arrivals are distributed uniformly.

The simulation algorithm is presented below. I assume that the agent needs to buy A units in a time period T and submits orders of the form (L, p, v, t) which indicates a limit buy order at price p for amount v at time t or (M, v, t) which indicates a market buy order. At the end of the time period, if A units have not been bought, then the trader makes a market order for the remainder of the inventory. I simulate updates in intervals of length b.

Algorithm 2: Simulation of Order Book

```
$n = 0$;
Let $L$ be the initial LOB at time 0. let $L_k$, $k = -K, \dots, -1, 1, \dots, K$ be the
  volumes at each position ;
Let $a$ be the amount of inventory filled ;
Let $P$ be the total price ;
while $n * b <= T$ and $a < A$ do
    Let $O$ be the orders submitted by the agent within the interval
      $[n * b, (n + 1) * b)$ ;
    Let $N_k^+, N_k^-$, $k = -K, \dots, -1, 1, \dots, K$ be the number of positive and
      negative arrivals at position $k$. Simulate these values using the procedure
      described in the beginning of the subsection ;
    For each $k$, generate $N_k^+$ uniform random variables $U \in [0, 1)$. The positive
      updates at position $i$ then occur at times $n * b + b * U$. Do the same for
      negative arrivals ;
    Take each update and order sorted by time. Treat each of the agent's order
      as an arrival. A market order or limit order above the best ask is executed
      immediately at the best price. A limit order is added to the list of active
      limit orders. Process updates as described in subsection 6.1.1. If a market
      order or active limit order is filled (must be from market order on the best
      bid price and taking into account the time priority policy), record the price
      and volume at which the order is executed and set $a \leftarrow a + v$, $P \leftarrow P + p$ ;
    $n = n + 1$ ;
end
Execute a market order to buy the remaining $A - a$ shares ;
Return $P$ ;
```

Chapter 7

Analysis

7.1 Model Behaviour

In this section, I will discuss any interesting behaviours of the queuing model used in simulation.

7.2 Trade Execution Strategy Analysis

Here, I will provide background on trade execution strategies and evaluate performance in the simulated environment. I will analyze the results and compare performances.

Chapter 8

Code Listings

In this section we include code listing for collecting and processing data. With this code, a user should be able to reproduce the experiment and run their own analysis. Code for the thesis can be found here: https://github.com/aw21/Thesis

8.1 Software Dependencies

Listing 8.1: 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
```

8.2 Data Collection

Listing 8.2: 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")
```

8.3 Data Processing

Listing 8.3: 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'])

    return starting_time, combined_updates

starting_time, combined_updates = processed_updates('12_28_18.json',10)
len(combined_updates)
```

8.4 Correlated Poisson Variables

Listing 8.4: Generating Correlated Poisson Variables using Copulas

```
import numpy as np
from scipy.stats import norm

n = 10000
y = [np.random.multivariate_normal(mean=[0,0],cov=[[1,-0.1],[-0.1,1]])
    ↪ for _ in range(n)]
# Correlated multivariate random variables
y1,y2 = zip(*y)

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
```

```
lambda1 = 2
lambda2 = 3

# Poisson random variables
x1 = [inverse_PDF_Poisson(lambda1,norm.cdf(y)) for y in y1]
x2 = [inverse_PDF_Poisson(lambda2,norm.cdf(y)) for y in y2]

print("Correlation Matrix:")
np.corrcoef(x1,x2)

print("Mean of x1: {}".format(np.mean(x1)))
print("Mean of x2: {}".format(np.mean(x2)))
```

Chapter 9

Conclusion

Here, I will summarize the experiment, results, and performance of the model.

9.1 Future Work

Here, I will discuss future work that could improve a part of the model (simulating the LOB, overcoming limitations, etc.) and how this work could be applied to gain additional insights.

Bibliography

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

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

Coinbase (2018). Coinbase markets trading rules. `https://www.coinbase.com/legal/trading_rules?locale=en-US`. Accessed: January 26, 2019.

CoinMarketCap (2018). Coinbase pro market cap. `https://coinmarketcap.com/exchanges/coinbase-pro/`. Accessed: January 26, 2019.

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.

Foundation, P. S. (2018). Copra. `https://pypi.org/project/copra/`. Accessed: January 26, 2019.

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*, page 110.

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.

Laruelle, S., Lehalle, C.-A., and Pags, G. (2013). Optimal posting price of limit orders: Learning by trading. *Mathematics and Financial Economics*, 7:359–403.

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

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.

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