Using Reinforcement Learning With Simulated Limit Order Book Data To Optimize Trade Execution: Fall Term 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

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 has allowed traders to infer information from limit order book (LOB) data and execute trades based on this information. In this paper, we attempt to solve the problem of optimal trade execution using reinforcement learning in a simulated market. Given the current state of the limit order book for a particular asset and the amount to buy (or sell), we model the problem as a Markov decision process and output the orders to place at each time step. Reinforcement learning has recently been used to optimize various agents' decisions given LOB data. However, it has relied on historical LOB data, so it has not accounted for the market impact on LOB dynamics from previous actions. To my knowledge, this paper is the first attempt of its kind to use reinforcement learning while updating LOB dynamics in real time. We first use high frequency LOB data to simulate LOB dynamics with real-time market impact adjustments using the queue reactive model developed by Huang et al. (2013). We then use reinforcement learning in the vein of Nevmyvaka et al. (2006) to model optimal trade decisions at each time period. We also explore various techniques to optimize performance of our model. Finally, we evaluate the performance of our model on testing data and compare it to common trade execution strategies. The findings from this paper are relevant for institutions who want to execute trades while maintaining profit as well as those who want to evaluate the effectiveness of reinforcement learning for trading decisions.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 Problem Description

This thesis attempts to solve 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, we 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, we 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 large institutional investors, 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.

1.2 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 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 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 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 by 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. We 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 2

Preliminary Literature Review

2.1 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. This thesis hopes to expand on this paper, whose methodology is briefly 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.

Although studies using reinforcement learning on LOBs have been conducted in the past (Nevmyvaka et al. (2006), Spooner et al. (2018)), neither included the impact of the agents trades on the future dynamics of the LOB. This thesis seeks to use reinforcement learning to solve the 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 based on queue reactive model of the LOB by Huang et al. (2013), which is described in Section 2.2. We can then build the reinforcement learning model within this simulated environment.

2.2 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, $[Qi : i = -1 \ldots -K]$ represents the number of orders on the bid side $i - 0.5$ ticks to the left of p_{ref} and $[Qi : i = 1 \ldots K]$ represents the number of orders on the ask side $i - 0.5$ 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 i occurs any time that the number of shares offered at position i 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.

For simplicity, we for now assume that Q_i is independent for each i. Each jump when Q_i increases by 1 and $Q = q$ follows a Poisson arrival with rate $\lambda_i(q)$ and each jump when Q_i decreases by 1 and $Q = q$ follows a Poisson arrival with rate $\gamma_i(q)$. For now, we might assume that the arrival rates at each i are independent of the size of the queue at any other i, but we may explore more complicated formulations in the course of the thesis. In order to model each Q_i as an integer with changes equal to 1, we use Q_i to represent the number of Average Event Sizes (AES_i) 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_i could differ. In this model, $Q_i = n$ would mean that there is $n * AESi$ shares offered at position i.

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

$$p = (b + a)/2$$

If b and a are an even number of ticks apart, then we have

$$p = (b + a)/2 + 0.5$$

Note that picking the reference price to be closer to the ask side is arbitrary. We have chosen this now for simplicity, but may revise it later to something more complicated (perhaps the side closest to the last 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_i, $\lambda_i(q)$, and $\gamma_i(q)$ for each i.

Chapter 3

Data Source

3.1 Coinbase Exchange

To simulate a real world security, we 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)). We 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 we model in this paper may not entirely reflect those of securities from other exchanges such as stocks, futures, etc., we 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)).

Chapter 4

Experimental Procedure

4.1 Simulating the LOB

4.1.1 Example Code

We present code written that is relevant for this thesis to demonstrate use of the Coinbase Pro API. Development was done in Python using a Jupyter notebook. We use of CoPrA, an asynchronous web socket client for Coinbase Pro (Foundation (2018)), to obtain LOB data with real time updates. The Coinbase Pro API provides us with a snapshot of LOB data that includes the price and volume of every bid and ask for an asset (consolidated so that orders of the same price are combined). By opening up a web socket, we also receive updates of all trades at each position of the LOB. The following code shows preliminary work using the API to model the dynamics of the ETC-USD orderbook, which is the exchange of Ethereum Classic for USD (around $2 million volume of trades each day). We start with the snapshot of the LOB to create our own internal representation of the LOB. We then update the order book whenever we receive an event from the API (which consists of a timestamp, price, and new volume). We update the order book with every event, and also calculate the AES for the first 5 bids and asks ($K = 5$ in the queue reactive model from above).

Although the LOB further extends on both sides, we assume that the majority of trades occur in the first K bids and asks. The code could also be adapted to estimate $\lambda_i(q)$, and $\gamma_i(q)$ for each i by using the time stamps of the events. The code is displayed in Listing 4.1:

Listing 4.1: Receiving LOB Data from Coinbase Pro and Calculating AES

```
import asyncio
from copra.websocket import Channel, Client
import matplotlib.pyplot as plt
from collections import OrderedDict

class Level2(Client):
    K=5
    Bids = {} # Prices and volumes of bids
    Asks = {} # Prices and volumes of asks
    reference_price = None
    first_bid = None # Best Bid
    first_ask = None # Best Ask

    total_event_sizes = OrderedDict([(i,0) for i in range(-K,K+1) if i
        ↪ != 0])
    numbers_of_events = OrderedDict([(i,0) for i in range(-K,K+1) if i
        ↪ != 0])

    # Displays AES at each i
    def display_AES(self):
        print("Average Event Sizes (AES):")
        for i in range(-self.K, self.K+1):
            if i != 0:
                total_event_size = self.total_event_sizes[i]
                n = self.numbers_of_events[i]
                if n == 0:
```

```
                AES = 0
            else:
                AES = total_event_size / n
            print("Level: {}, n = {}, AES: {}".format(i, n, AES))
        print("\n")

    # Displays Order Book
    def display_order_book(self):
        bids = self.Bids.items()
        bids = [b for b in bids if (self.reference_price - b[0]) < self.
            ↪ K]
        bid_prices = [b[0] for b in bids]
        bid_volumes = [b[1] for b in bids]

        asks = self.Asks.items()
        asks = [a for a in asks if (a[0] - self.reference_price) < self.
            ↪ K]
        ask_prices = [a[0] for a in asks]
        ask_volumes = [a[1] for a in asks]

        b1 = plt.bar(bid_prices, bid_volumes, color='r')
        b2 = plt.bar(ask_prices, ask_volumes, color='g')
        v = plt.axvline(x=self.reference_price, color='b')
        plt.xticks([x for x in range(min(bid_prices), max(ask_prices)+1)
            ↪ ])

        plt.title('ETC-USD Limit Order Book: Depth = {}'.format(self.K))
        plt.legend(['Reference Price = {}'.format(self.reference_price),
            ↪  'Bids', 'Asks'])
        plt.xlabel('Price (USD Cents)')
        plt.ylabel('Volume')
        plt.show()
```

```
# Updates the reference price of order book
def update_order_book(self):
    sorted_bids = list(reversed(sorted(self.Bids.items())))
    sorted_asks = list(sorted(self.Asks.items()))
    best_bid = sorted_bids[0]
    best_ask = sorted_asks[0]
    if ((best_bid[0] + best_ask[0]) % 2) != 0:
        self.reference_price = round((best_bid[0]+best_ask[0])/2, 1)
    else:
        self.reference_price = round((best_bid[0]+best_ask[0])/2 +
            ↪ 0.5,1)

    self.first_bid = int(round(self.reference_price - 0.5))
    self.first_ask = int(round(self.reference_price + 0.5))

# Prints LOB
def print_order_book(self):
    print("Reference Price: {}".format(self.reference_price))
    print("----------------------------------------")

    print("First {} Bids: ".format(self.K))
    for price in range(self.first_bid - self.K + 1, self.first_bid +
        ↪ 1):
        print("{}, {}".format(price, self.Bids.get(price, 0)))
    print("----------------------------------------")

    print("First {} Asks: ".format(self.K))
    for price in range(self.first_ask, self.first_ask + self.K):
        print("{}, {}".format(price, self.Asks.get(price, 0)))
    print("----------------------------------------")
```

```
# Receives message from API websocket
def on_message(self, message):
    # Get snapshot of LOB from API and build internal representation
    if message['type'] == 'snapshot':
        for price, amount in message['bids']:
            self.Bids[int(round((float(price)*100)))] = float(amount
                ↪ )
        for price, amount in message['asks']:
            self.Asks[int(round((float(price)*100)))] = float(amount
                ↪ )
        self.update_order_book()
        self.display_order_book()
        self.print_order_book()

    # Update order book when new event occurs
    if message['type'] == 'l2update' and 'time' in message:
        significant_order = False
        for (side, price, amount) in message['changes']:
            print("Update: {}".format((side, price, amount)))
            print("Time: {}".format(message['time']))
            print("----------------------------------------")

            # Find i from the price. Keep track of event if
            # abs(i) <= K
            price = int(round((float(price)*100)))
            i = price - self.reference_price
            if i < 0:
                i = int(round(i - 0.5))
            else:
                i = int(round(i + 0.5))
            if abs(i) <= self.K:
                if i < 0:
```

```
                    change = float(amount) - self.Bids[price]
                else:
                    change = float(amount) - self.Asks[price]
                self.total_event_sizes[i] += abs(change)
                self.numbers_of_events[i] += 1
                significant_order = True

            # Update volume in order book
            if side == "buy":
                if amount == "0":
                    del self.Bids[price]
                else:
                    self.Bids[price] = float(amount)
            elif side == "sell":
                if amount == "0":
                    del self.Asks[price]
                else:
                    self.Asks[price] = float(amount)

        # Update reference price if needed
        self.update_order_book()

        if significant_order:
            self.print_order_book()
            self.display_AES()



loop = asyncio.get_event_loop()
channel = Channel('level2', 'ETC-USD')
level2data = Level2(loop, channel)
```

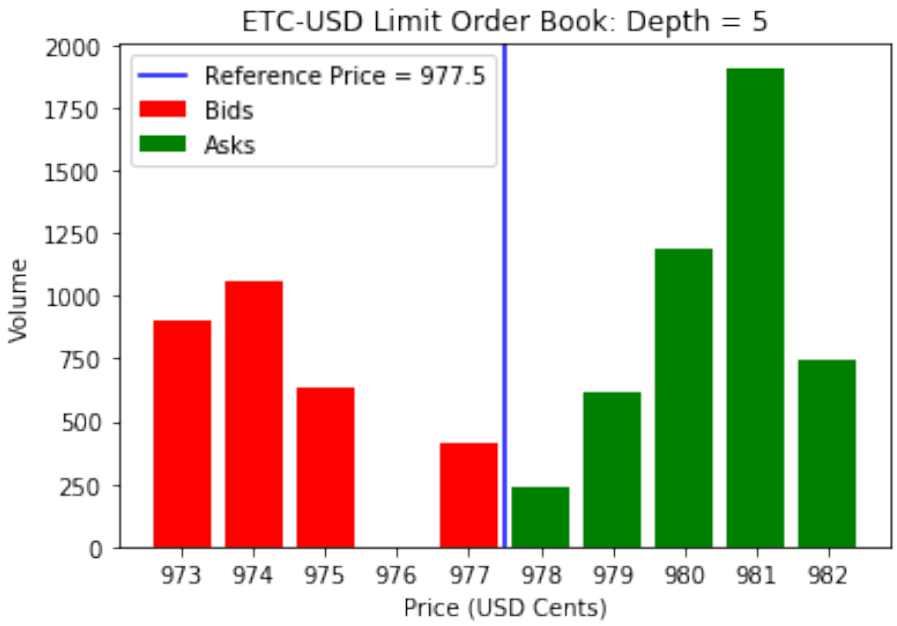

Figure 4.1: LOB Sample

```
try:
    loop.run_forever()
except KeyboardInterrupt:
    loop.run_until_complete(ws.close())
    loop.close()
```

Sample output is shown in Figure 4.1 and Listing 4.2. This includes the plot of the LOB from when we take the first snapshot as well as the first update.

Listing 4.2: Processing Next Order and Updating LOB

```
Reference Price: 977.5
-----------------------------------------

First 5 Bids:
973, 900.08
974, 1057.308
975, 634.83242699
976, 0.5
977, 411.46
```

```
First 5 Asks:
978, 239.18047425
979, 612.21451217
980, 1182.36704901
981, 1908.3268
982, 740.28201
```

```
Update: ('buy', '9.77000000', '258.16')
Time: 2018-11-07T04:33:44.032Z
```

```
Reference Price: 977.5
```

```
First 5 Bids:
973, 900.08
974, 1057.308
975, 634.83242699
976, 0.5
977, 258.16
```

```
First 5 Asks:
978, 239.18047425
979, 612.21451217
980, 1182.36704901
981, 1908.3268
982, 740.28201
```

```
Average Event Sizes (AES):
Level: -5, n = 0, AES: 0
Level: -4, n = 0, AES: 0
Level: -3, n = 0, AES: 0
Level: -2, n = 0, AES: 0
```

```
Level: -1, n = 1, AES: 153.29999999999995
Level: 1, n = 0, AES: 0
Level: 2, n = 0, AES: 0
Level: 3, n = 0, AES: 0
Level: 4, n = 0, AES: 0
Level: 5, n = 0, AES: 0
```

After 3 minutes, we have the following order book and AESs (Listing 4.3)

Listing 4.3: Updated LOB after 3 Minutes

```
Update: ('buy', '9.70000000', '659.83392355')
Time: 2018-11-07T04:36:40.092Z
--------------------------------------------------
Reference Price: 974.5
--------------------------------------------------
First 5 Bids:
970, 659.83392355
971, 0
972, 0
973, 0
974, 0
--------------------------------------------------
First 5 Asks:
975, 0
976, 0
977, 0
978, 451.62338343
979, 607.27924933
--------------------------------------------------
Average Event Sizes (AES):
Level: -5, n = 21, AES: 100.37783655666668
Level: -4, n = 18, AES: 179.98507636055558
Level: -3, n = 7, AES: 233.21600000000004
```

```
Level: -2, n = 6, AES: 106.23010864166667
Level: -1, n = 12, AES: 47.03833333333333
Level: 1, n = 10, AES: 47.995709082000005
Level: 2, n = 4, AES: 158.94729133999996
Level: 3, n = 16, AES: 117.74859454
Level: 4, n = 30, AES: 234.910389792
Level: 5, n = 16, AES: 313.808875
```

As can be seen, the reference price went from \$9.775 to \$9.745, possibly due to large taker order from the ask side. For the actual thesis work, we would run the code for a much longer (possibly multiple days) time to estimate AES, $\lambda_(q)$, and $\gamma_i(q)$ for each i.

4.1.2 Using Parameters to Simulate LOB

Here, I will use the values that I estimated from above to write a program that simulates the LOB using the queue reactive model.

4.2 Reinforcement Learning Model

4.2.1 Reinforcement Learning Background

Here, I will provide background on reinforcement learning that is necessary to understand the experiment and why it is a good choice for this problem.

4.2.2 Machine Learning Architecture

Here, I will describe the programming environment and machine learning model for the problem.

Chapter 5

Analysis

5.1 Experimental Results

Here, I will present the results of the experiment and analyze the output. I will discuss performance of the model any insights received from the experiment.

5.2 Model Evaluation

Here, I will compare the performance of the model to other common methods of optimal trade execution.

5.3 Feasibility of Model

Here, I will discuss the feasibility of the model on other real-world data-sets. I will discuss how realistic the queue reactive model is for modelling LOB dynamics, as well as how well I expect the reinforcement learning model to generalize to other data-sets. I will then discuss any limitations that I had when working on the thesis and whether they could be overcome.

Chapter 6

Timeline

6.1 Summary of Major Work that Needs to be Done

- Update the above code to estimate the Poisson arrival rates of events for the model.
- Use the values of AES, $\lambda_i(q)$, and $\gamma_i(q)$ to build a market simulation that allows for updates in real time.
- Build a reinforcement learning model for optimal trading execution that uses the environment from the simulation above.
- Compare performance to benchmark strategies and for multiple currencies.
- Write the thesis in LaTeX.

6.2 Rough Timeline

- By the end of the semester in December, I hope to have code written that simulates the market using the queue reactive model.

- By January, I hope to have preliminary code running for a reinforcement learning model based on this simulation.
- By February, I hope to have revised the code, tweaked hyperparameters, collected enough data, etc. to evaluate the performance of my model with meaningful results to write my thesis.
- By the April deadline, I really hope to have my thesis written.

Chapter 7

Conclusion

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

7.1 Future Work

Here, I will discuss future work that could improve a part of the model (simulating the LOB, the machine learning mode, 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.

Coinbase (2018). Coinbase markets trading rules. `https://www.coinbase.com/legal/trading_rules?locale=en-US`. Accessed: November 9, 2018.

CoinMarketCap (2018). Coinbase pro market cap. `https://coinmarketcap.com/exchanges/coinbase-pro/`. Accessed: November 9, 2018.

Foundation, P. S. (2018). Copra. `https://pypi.org/project/copra/`. Accessed: November 9, 2018.

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.

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.

Spooner, T., Fearnley, J., Savani, R., and Koukorinis, A. (2018). Market making via reinforcement learning. *CoRR*, abs/1804.04216.