IMPERIAL COLLEGE LONDON

APPLIED PROJECT

Optimizing Market Making Strategies via Inventory Management and Order Book Analysis

Author:
Talha Jamal

CID: 01357497

An Applied Project submitted in fulfillment of the requirements for the degree of Master's of Science in Risk Management & Financial Engineering

in the

Financial Engineering Master's
Imperial College London

August 26, 2024

Client Specification

R^2 Markets is a market maker involved within Crypto Spot and Options Markets, and is looking to build a Quantitative Model that can help them provide liquidity within the crypto spot market. With a goal to provide liquidity in an asset class that is extremely volatile, R^2 Markets rely on advanced statistical analysis to make markets in crypto spot and derivatives to exchange trading requirements whilst also maintaining profitability.

It is important for R^2 Markets to find the optimal fair value of an asset whilst it is trading assets electronically over an exchange. Constantly looking at Order Book statistics and managing the firm's inventory is necessary for the Market Maker to find it's own fair value of an asset and to quote bids and offers at an appropriate spread. Skewing the prices shown on the exchange when an asset's inventory is too high or low with R^2 Markets is important to increase the probability of directional trades that ensure R^2 Markets has a delta risk of 0 and makes profits regardless of the directional trend of the market.

Contents

Client Specification ii

1 Introduction 1
- 1.1 Electronic Markets 1
- 1.2 Market Microstructure 1
- 1.3 What is a Market Maker? 1
- 1.4 How do Market Makers make money? 1
- 1.5 Evolution of Market Makers 2
- 1.6 Challenges faced by Market Makers 2

2 Theoretical Background and Literature Review 3
- 2.1 Limit Order Books 3
- 2.2 Statistical Properties of a Limit Order Book 4
- 2.3 Inventory Management 4

3 Methodology 5
- 3.1 Model 5
- 3.2 Mid Price 5
- 3.3 Simulation of Prices 5
- 3.4 Optimal Bids and Asks 6
- 3.5 Optimal Trading Volume 7

4 Results and Discussion 9
- 4.1 Price Simulation 9
- 4.2 Limit Order Book 9
- 4.3 Market Maker Trading Behaviour 10
- 4.4 Conclusion 12

Bibliography 13

A Appendix 15

Chapter 1

Introduction

1.1 Electronic Markets

With the turn of the century, electronic markets have taken over the world. Automated Systems of Exchanges route orders between buyers and sellers of assets like stocks, bonds, currencies, commodities and derivatives on these underlying assets. This system has revolutionised the scale and speed at which trading now takes place across the globe.

1.2 Market Microstructure

Electronic Markets facilitate trading between participants by allowing them to buy and sell assets via orders posted on exchanges. The exchange reflects the prices - the bid and asks for each asset at any given time.

- **Bid:** The highest price a buyer is willing to pay for a security at any given moment.
- **Ask:** The lowest price a seller is willing to sell a security for at any given moment.

The spread for an asset is the difference between the best bid and best ask at any time. This spread represents the potential profit before accounting transaction costs for Market Makers if they are able to execute a two way trade: buying at the bid and selling at the ask.

1.3 What is a Market Maker?

Market Makers are financial intermediaries who provide liquidity to market participants by buying and selling securities electronically over an exchange. They quote two way prices on exchanges to trade bidirectionally leading to smoother trading for market participants, smaller bid/ask spreads for assets as Market Makers compete for orders with each other by posting competitive bids and asks, and lower trading costs for the market in general.

1.4 How do Market Makers make money?

Market Makers earn their profits from the spread of their quotes. If their internal model calculates 20$ to be the fair value of an asset and they can get a trade executed at 19.50 $ to buy the asset and 20.50 $ to sell the asset, they generate a 1$ spread for themselves before taking into account trading costs, slippage, and execution fees. They are hence inclined to always prefer high trading volumes to scale their profits by earning the spread on assets they trade.

1.5 Evolution of Market Makers

Before the advent of electronic markets, market makers used to be smaller firms or individuals quoting prices to trade on the floors of Stock Exchanges. However, with the advent of electronic markets, computing technologies, and the internet, algorithmic trading has flourished and given individuals and firms the capability to trade at a much higher scale than ever before by trading at higher speed and volumes.

1.6 Challenges faced by Market Makers

As competition has grown within the industry, market makers have competed for the traded volume of exchanges that flow through their orders. This has led to the total spread - the difference between the bid and ask of an asset - decreasing over time. To stay at the top of the order book, market makers need to constantly improve their physical trading hardware, their quantitative models to generate fair values of assets, and their algorithms that trade electronically on exchanges globally. Some of the challenges faced by market makers are listed below:

- Managing risk of holding inventory in volatile environments.
- Managing regulatory scrutiny and meeting their trading volume requirements as per their agreements with exchanges.
- Building Quantitative Models to build fair values of assets they trade.
- Building and maintaining complex software and hardware to trade assets electronically.
- Delta Risk of the price of an asset moving against them while they have it within their inventories.
- Maintaining accurate models that adjust their orders and pricing quotes within volatile trading environments.

Chapter 2

Theoretical Background and Literature Review

2.1 Limit Order Books

Limit Order Books aggregate the outstanding buy and sell orders posted by market participants. They specify the price and volume a market participant is willing to buy or sell a particular security. It provides transparency to electronic markets and matches the best buy and best ask so that transactions can take place within the marketplace. Here are the characteristics of a limit order book:

1. **Best Bid:** The highest price a buyer is willing to buy a security at.
2. **Best Ask:** The lowest price a seller is willing to sell a security for.
3. **Order Depth:** The depth of a traded security is measured as the number of different orders at different price levels.
4. **Order Priority:** Different exchanges and order books give priority to different orders in terms of execution. Majority execute the order with the best price first, and then execute orders of the same price with a first come first served rule.
5. **Liquidity:** The deeper an order book for a security, the more liquid that security as it's orders can get executed at deeper price levels.

Table 2.1 below illustrates an example of a Limit Order Book reflecting the different orders places by market participants for a given security.

Price	Volume (Bids)	Volume (Asks)
100.5	-	200
100.4	-	150
100.3	-	100
100.2	100	-
100.1	150	-
100.0	200	-

TABLE 2.1: Example of a Limit Order Book

2.2 Statistical Properties of a Limit Order Book

Bouchaud, et.al. [1] study the statistical properties of a limit order book and discovered that it can help identify the market dynamics and price formation process for securities in a given market. Analysis of limit order books can thus help in analyzing order flow, which can then be used to make informed decisions on the price and fair value of a security.

With the advent of technology and abundant data available about markets, it is easy to reconstruct limit order books by looking at publicly available data released by exchanges. By reconstructing a limit order book, one can analyze the below static properties of a limit order book :

- Distribution of incoming limit orders [1].
- Average shape of the order book [1].
- Distribution of volume at the bid/ask [1].
- Order book depth [1].

2.3 Inventory Management

Alongside the analysis of a limit order book, the key to a market maker generating appropriate quotes for a security is management of it's own inventory. Avellaneda and Stoikov [2] identify the two biggest risk to a Market Maker:

- Inventory Risk arising from uncertainty in the asset's value [2].
- Assymetric information risk arising from informed investors [2].

Avellaneda and Stoikov's paper [2] develops the idea that a Market Maker can skew it's bid/ask quotes based on the amount of inventory it either already has of the asset, or the inventory it needs from the market. Skewing it's big/ask quotes according to inventory management ensures the market maker keeps it's delta risk (risk of the price of the asset moving against it while it keeps the asset in its inventory) close to 0 by getting rid of any asset in it's inventory.

If the market maker has a large amount of inventory of a security it wants to get rid of, it can decrease it's ask so it's limit orders get executed and it can sell these assets. Simultaneously, if the Market Maker needs to source a security, it can increase it's bid to ensure it's limit orders on the exchange get executed and the Market Maker can buy the asset. Enabling the market maker's model to skew prices as a function of the amount of inventory it has reduces the market maker's inventory risk.

Chapter 3

Methodology

3.1 Model

To numerically simulate a market maker's role within an Electronic Trading Marketplace, 1000s of uninformed betting agents were created to generate buy and sell orders at each given time. The betting agents trade the single asset at a coin flip: the probability of going long or short the asset is the same for each agent. The simulation of their orders synthetically creates a marketplace for the single asset, with buyers quoting bids and sellers quoting asks. A Limit Order Book was created to aggregate these orders by price and volume and execute matching orders by time priority first and then on a volume priority basis.

A market maker agent was created to make a market for this single asset by quoting both ways for the asset. The market maker uses an inventory model similar to [2] where the market maker's optimal bids and asks are skewed depending on the amount of inventory it holds. The market maker quotes optimal volume for its bids and asks by looking at order book statistics similar to [1], such as order book depth, for optimal execution.

3.2 Mid Price

For this research, the fair value of the asset is assumed to be the mid price of the asset. Consider $a(t)$ as the ask price of an asset at time t and $b(t)$ as the bid price of an asset at time t. It's mid price $m(t)$ calculated as showing in equation 3.1.

$$m(t) = \frac{a(t) + b(t)}{2} \tag{3.1}$$

3.3 Simulation of Prices

For our simulation, we use the following model to simulate m_t:

- **Brownian Motion:** The model simulates $m(t)$ in equation 3.1 evolving according to a stochastic process 3.2 where S_t reflects the Stock Price, W_t is a standard 1 dimensional Brownian Motion, and σ volatility of the asset is constant. The change in the price here follows a simple Brownian Motion with a drift of zero. The change in S_t is directly proportional to the random component dW_u. This change is also additive and remains a linear function of the Brownian Motion. The SDE in 3.2 is numerically simulated via the Euler Maruyama method and can be visualized in Figure 3.1 below.

$$dS_u = \sigma \, dW_u \tag{3.2}$$

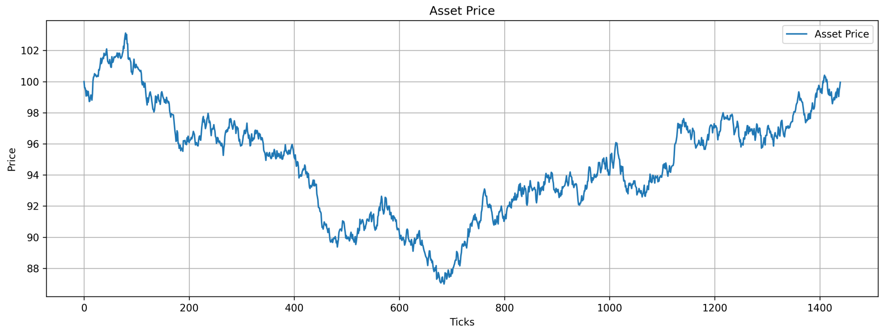

FIGURE 3.1: Simulation of Asset Price

3.4 Optimal Bids and Asks

The optimal bid and ask prices are calculated using the Avellaneda-Stoikov model [2].

Reservation Price

The market maker calculates a reservation price 3.3 for the asset. This is the price where the market maker is indifferent to the asset - it is the market maker's optimal price for the asset given the market maker's current inventory, asset's volatility, market maker's risk aversion, and the time horizon for the strategy. The market maker's bids and asks are quoted around the reservation price.

$$r_t = S_t - \gamma \cdot I \cdot \sigma^2 \cdot T \tag{3.3}$$

Bid Price

The market maker's bid is below it's reservation price 3.3. The shift below the reservation price is a function of the market maker's risk aversion and market impact. With a higher γ the bid price decreases because the risk-averse market maker wants to lower the buying price to manage risk. A higher κ(lower market impact) results in a smaller adjustment, tightening the bid price closer to the reservation price. A lower κ (higher market impact) results in a larger downward adjustment, reducing the bid price further from the reservation price. Using a logarithmic shift allows the model add a non-linear shift and embed a concave risk-averse utility function into the pricing strategy.

$$p_{bid} = r_t - \frac{1}{\gamma} \cdot \log\left(1 + \frac{\gamma}{\kappa}\right) \tag{3.4}$$

Ask Price

The ask price is adjusted above the reservation price 3.3. A higher γ (more risk-averse) results in a smaller adjustment term, leading to a more conservative (higher) ask price. A higher κ (lower market impact) results in a smaller upward adjustment, bringing the ask price closer to the reservation price and vice versa. Using a logarithmic shift allows the model to add a non-linear shift and embed a concave risk-averse utility function into the pricing strategy.

$$p_{ask} = r_t + \frac{1}{\gamma} \cdot \log\left(1 + \frac{\gamma}{\kappa}\right) \tag{3.5}$$

where:

- S_t: Current mid-price of the asset.
- I: Current inventory level. The current inventory influences the direction of the reservation price adjustment. If the market maker holds a large positive inventory, they will adjust the reservation price downward to reduce the incentive to buy more and to encourage selling. Conversely, a negative inventory (short position) will increase the reservation price to discourage further selling and encourage buying.
- σ: Volatility of the asset. Higher volatility increases the potential risk associated with the inventory, leading to a larger adjustment in the reservation price. This makes the market maker more cautious, shifting the reservation price away from the mid-price more significantly.
- T: Time horizon for the market maker. A longer time horizon increases the influence of the inventory on the reservation price. This reflects the idea that the longer the market maker plans to hold their position, the more cautious they need to be about accumulating excessive inventory, hence a greater adjustment to the reservation price.
- γ: Risk aversion parameter. A higher γ leads to more conservative pricing (wider spreads) and lower trading volumes. It effectively controls how aggressively the market maker is willing to trade and how much inventory risk they are willing to take.
- κ: Market impact parameter. Affects the spread of the quotes. Lower market impact (higher κ) allows for tighter spreads, which can make the market maker more competitive, but also increases the risk of adverse selection.

3.5 Optimal Trading Volume

The optimal trading volume is calculated based on several factors:

Inventory Factor

This factor adjusts the volume based on the market maker's current inventory. If the inventory is large (either positive or negative), the factor reduces, leading to smaller trading volumes to avoid increasing exposure. Conversely, smaller inventories lead to higher trading volumes.

$$\text{Inventory Factor} = \max\left(0.5, 1 - 0.5 \cdot \theta \cdot |I|\right) \tag{3.6}$$

Order Book Depth Factor

Represents the market depth, which is the overall availability of buy and sell orders in the order book. A higher depth suggests more liquidity, allowing larger trades without significantly impacting the price. The factor scales with the average volume in the order book but is capped to prevent excessively large trades.

$$\text{Depth Factor} = \min\left(2, \frac{\mu(\text{All Volumes})}{5000}\right) \tag{3.7}$$

Wealth Factor

This factor adjusts volume based on the market maker's cash reserves relative to the value of the inventory. If the cash reserves are low relative to the inventory value, the factor decreases, leading to smaller trade sizes. The wealth factor also accounts for the market maker's risk aversion, reducing volumes as risk aversion increases.

$$\text{Wealth Factor} = \max\left(0.5, \frac{C}{C + I \cdot S_t}\right) \cdot (1 - \theta) \tag{3.8}$$

Optimal Volume

$$V_{opt} = \text{Inventory Factor} \cdot \text{Depth Factor} \cdot \text{Wealth Factor} \cdot 10^4 \tag{3.9}$$

where:

- I: Current inventory level.
- θ: Risk aversion parameter.
- $\mu(\text{All Volumes})$: Mean of the combined buy and sell volumes in the order book.
- C: Current cash balance.
- S_t: Current mid-price of the asset.
- V_{opt}: Optimal trading volume.

Chapter 4

Results and Discussion

4.1 Price Simulation

A single asset's midprice $m(t)$ 3.1 is simulated for 3 trading days with tick by tick data every minute. Each day has 8 hours of trading. The model does not trade the first day as it needs pricing data to calculate input parameters such as σ, the volatility of the asset.

4.2 Limit Order Book

A limit order book was created to aggregate the buy and sell orders from 1000s of uninformed agents. Each uninformed agent has an aggression level sampled from a uniform distribution between [0, 0.4]. The higher the aggression level for the agent, the more competitive it's bid and ask and vice versa. Simultaneously, at every tick, the market maker was quoting bids and asks at it's calculated optimal quotes and volume. A snapshot of the order book can be visualized in Figure 4.1 below.

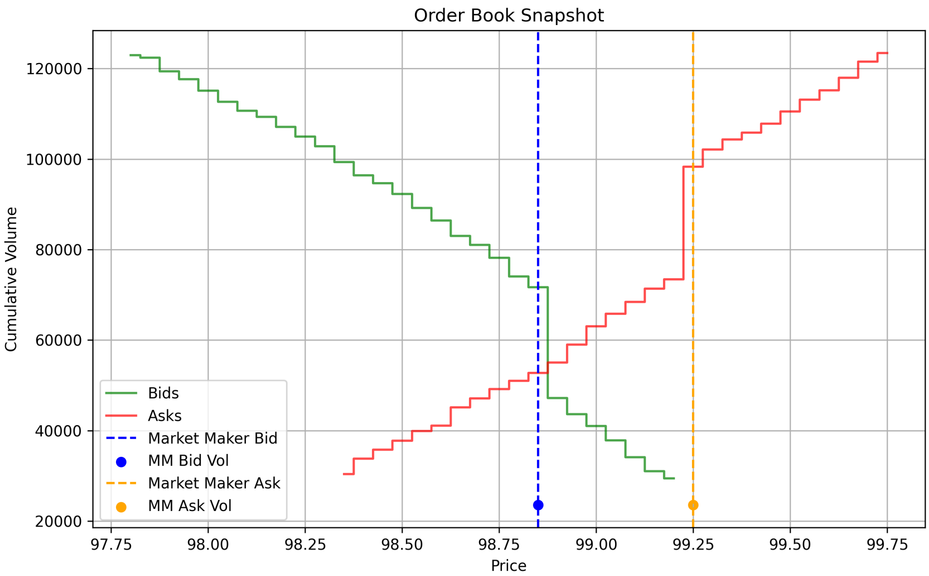

FIGURE 4.1: Order Book Snapshot

4.3 Market Maker Trading Behaviour

The Market Maker started with 10 million $ and 0 inventory of the asset. Over 2 days of trading activity, the market maker's PnL and Inventory over time were observed.

Market Maker PnL

The Market Maker's PnL was calculated by summing up it's realized and unrealized PnL.

$$\text{Realized PnL} = \sum_{i=1}^{N} \left(p_{\text{sell},i} - p_{\text{buy},i}\right) \times q_i \tag{4.1}$$

$$\text{Unrealized PnL} = \text{Inventory} \times \left(p_{\text{current}} - p_{\text{last trade}}\right) \tag{4.2}$$

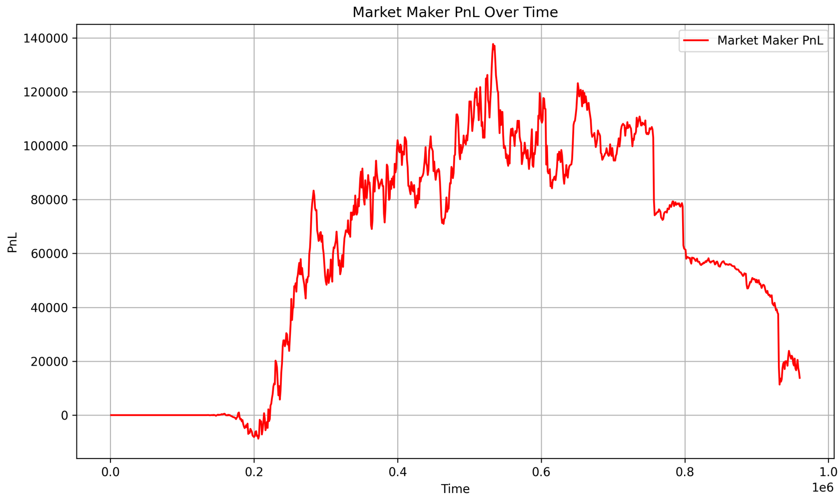

FIGURE 4.2: Market Maker PnL

The Market Maker's Inventory can also be visualized in the Figure 4.3 below.

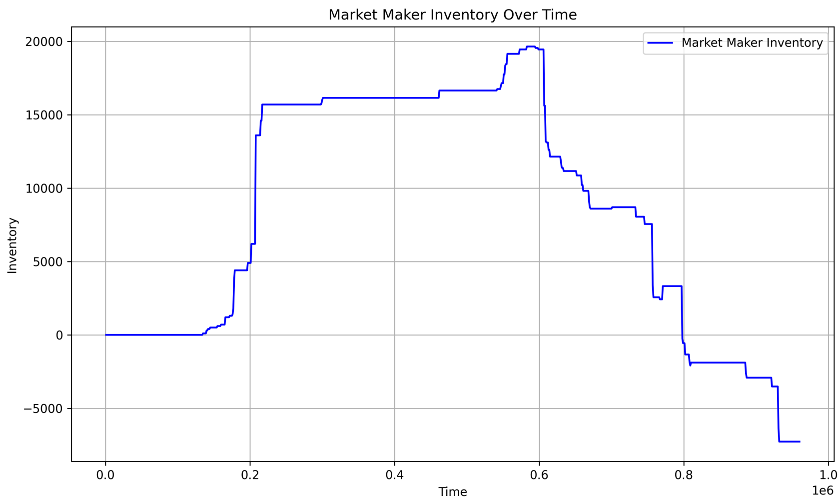

FIGURE 4.3: Market Maker's Inventory over time

For roughly first 180 ticks (3 hours) the market maker's was not able to buy any inventory. This was probably due to the market maker's quotes not being executed. Over time, as the market maker calculated it's fill rate and adjusted it's quotes to get filled, the market maker was able to accumulate inventory as it's orders got executed. This is when the market maker's PnL became positive as a result of trading activity as well. The objective, however, of the market maker is to always have close to 0 inventory over the long run. This can be observed here as well as after some time, the market maker started to aggressively reduce it's inventory. Perhaps during this time the market maker lost PnL as well as it aggressively shifted its quotes (reduced its ask) to sell the inventory as the market was ticking up.

Order Fill Rate

The Market Maker's fill rate was calculated via equation 4.3 at each trading time. If the market maker was not being filled on it's orders, the market maker's quotes were shifted in the next trading horizon to make it's quotes more likely to get filled. As mentioned in the previous section, it can be observed in the Figure 4.4 below that for the first 180 ticks the market maker's orders were not being filled. However, over time as the market maker adjusted it's quotes, the fill rate started to improve.

$$\text{Fill Rate} = \frac{\text{Orders Executed}}{\text{Total Orders Placed}} \times 100\% \tag{4.3}$$

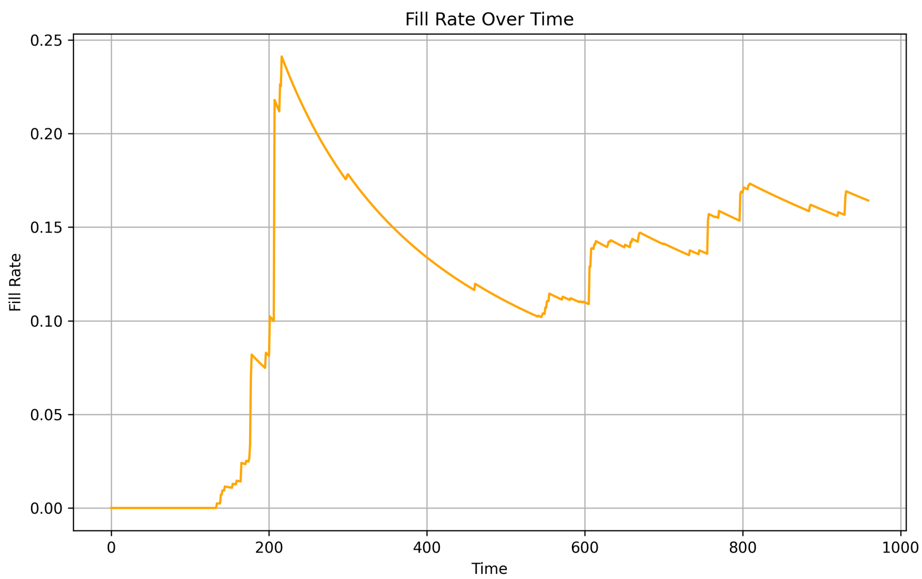

FIGURE 4.4: Market Maker's Fill Rate

Bid and Asks over time

At each tick the market maker's bids and asks were compared to the best bid and ask in the order book. At roughly 180 ticks when the market maker's bid were executed and the market maker was long inventory of the asset, the market maker correctly reduced it's bid (the downward spike can be observed in Figure 4.5) to decrease the likelihood of buying more inventory of the asset. This aligns with the market maker's strategy of avoiding accumulation of inventory. The Market Maker adjusts it's ask significantly upwards in an attempt to sell inventory at a favorable price. The distance to best ask increases as a result of the market maker's cautious selling strategy. The market maker's fill rate goes down as a result, as can be observed in Figure 4.4

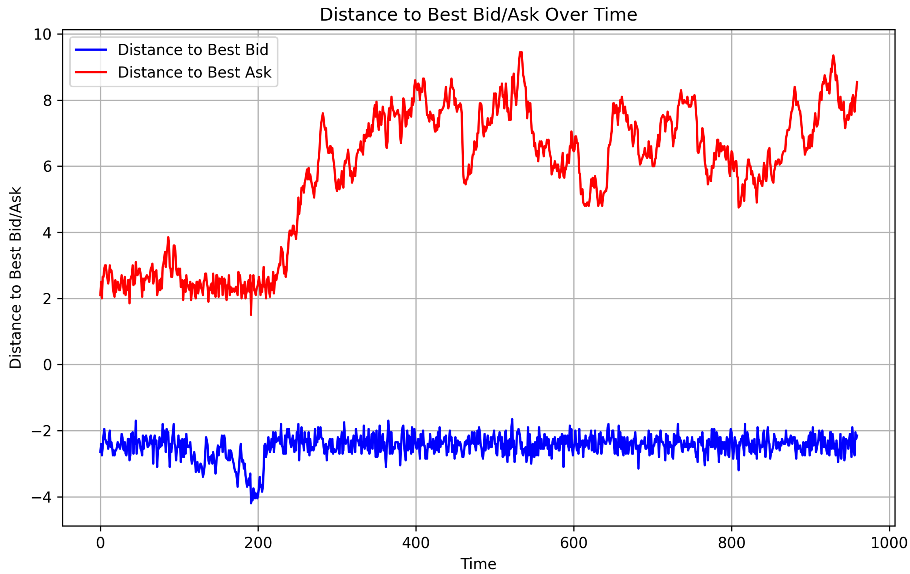

FIGURE 4.5: Distance to Best Bid and Best Ask

4.4 Conclusion

The Market Maker model inspired by Avellaneda and Stoikov [2] performs well overall and accumulates 20k $ PnL over the 2 trading days. However, the market maker model can improve on the following:

- The market maker does not have a fast enough feedback loop to improve it's quotes and volumes when it's orders are not being filled.
- The market maker does not trade in and out of positions fast enough.

In future work, the market maker's risk aversion and factors such as the inventory factor 3.6 and order book depth should be cross validated to find an optimal parameter number to optimize for the market maker's PnL.

Bibliography

[1] Jean-Philippe Bouchaud, Marc Mézard, and Marc Potters. Statistical properties of stock order books: empirical results and models. *Quantitative Finance*, 2(4):251–256, August 2002.

[2] Marco Avellaneda and Sasha Stoikov. High-frequency trading in a limit order book. *Quantitative Finance*, 8(3):217–224, 2008.

Appendix A

Appendix

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime, timedelta
import collections
import random
import sys
import warnings
warnings.filterwarnings('ignore')

# Price Simulation Class
class PriceSimulator:
    def __init__(self, days, initial_price=100, mu=0.0001, sigma=0.2, minutes_per_day=480, dt=1, seed=4):
        self.initial_price = initial_price
        self.mu = mu
        self.sigma = sigma
        self.days = days
        self.minutes_per_day = minutes_per_day
        self.T = self.days * self.minutes_per_day
        self.dt = dt
        self.seed = seed
        self.N = int(self.T / self.dt)
        self.brownian_motion_prices = np.zeros(self.N)
        self.price_df = pd.DataFrame()

    def simulate_brownian_motion_prices(self):
        np.random.seed(self.seed)
        self.brownian_motion_prices[0] = round(self.initial_price, 2)

        for t in range(1, self.N):
            Z_t = np.random.randn()
            self.brownian_motion_prices[t] = round(self.brownian_motion_prices[t-1] + \
                                                   self.sigma * np.sqrt(self.dt) * Z_t, 2)

        return self.brownian_motion_prices

    def generate_trading_days(self):
        self.start_date = datetime(2023, 1, 1)
        self.trading_days = []
        while len(self.trading_days) < self.days:
            if self.start_date.weekday() < 5:
                self.trading_days.append(self.start_date)
            self.start_date += timedelta(days=1)

    def generate_time_series(self):
        self.time_series = []
```

```
        for day in self.trading_days:
            for minute in range(self.minutes_per_day):
                self.time_series.append(day + timedelta(minutes=minute)
    )

    def create_dataframe(self, prices):
        self.generate_trading_days()
        self.generate_time_series()

        if len(prices) != len(self.time_series):
            raise ValueError(f"Mismatch in lengths: Prices({len(prices)
    }) vs Time Series({len(self.time_series)})")

        self.price_df = pd.DataFrame({
            'datetime': self.time_series,
            'price': prices
        })

        self.price_df['date'] = self.price_df['datetime'].dt.strftime('
    %Y-%m-%d')
        self.price_df['time'] = self.price_df['datetime'].dt.time
        self.price_df.drop(columns=['datetime'], inplace=True)

        self.price_df['tick_by_tick_return'] = self.price_df['price'].
    pct_change().fillna(0)
        self.price_df = self.price_df[["date", "time", "price", "
    tick_by_tick_return"]]
        self.price_df.to_csv("data/pricing_data.csv", index=False)
        return self.price_df

    def plot_prices(self, figsize:tuple=(15, 5)):
        plt.figure(figsize=figsize)
        plt.plot(self.price_df["price"], label="Asset Price")
        plt.xlabel("Ticks")
        plt.ylabel("Price")
        plt.title("Asset Price")
        plt.legend()
        plt.grid(visible=True)
        plt.savefig("data/price_simulation.png", dpi=300)

# Agent Classes
class Agent:
    def __init__(self, id, initial_cash, initial_inventory,
    aggressiveness):
        self.id = id
        self.cash = initial_cash
        self.inventory = initial_inventory
        self.aggressiveness = aggressiveness
        self.order_history = []

class UninformedInvestorAgent(Agent):
    def __init__(self, id, initial_cash, initial_inventory,
    aggressiveness):
        super().__init__(id, initial_cash, initial_inventory,
    aggressiveness)
        self.momentum_period = random.choice([1, 2, 3, 4, 5])  # Choose
     a random momentum period between 1 and 5 days
        self.momentum_tag = f"{self.momentum_period}-day Momentum"
        self.current_position = None  # Track current open position
        self.hold_time = None  # To store the holding period
        self.last_trade_time = None  # To track when the position was
    opened

```

```python
        # DataFrame to track orders, cash, PnL, etc.
        self.metrics_df = pd.DataFrame(columns=[
            "momentum_trader", "aggressiveness", "timestamp", "cash",
            "inventory", "pnl", "order_type", "order_price", "order_size",
            "order_dollar_value", "position"
        ])
        self.pnl = 0  # Initialize PnL

    def round_to_tick(self, price):
        """Round price to the nearest 0.05 increment."""
        return round(price * 20) / 20.0

    def place_order(self, recent_prices, current_time):
        # Generate a random direction and price deviation
        direction = np.random.choice(['buy', 'sell'])
        price_deviation = np.random.uniform(-0.01, 0.01)
        price = self.round_to_tick(recent_prices[-1] * (1 + price_deviation))

        # Generate order size
        order_size = round((self.cash * self.aggressiveness) / price, 2)

        # Ensure that the order is valid
        if direction == 'buy':
            bid_price = price
            ask_price = self.round_to_tick(recent_prices[-1] * (1 + 0.005))  # Slightly above the current price

            if bid_price >= ask_price:
                bid_price = self.round_to_tick(ask_price - 0.05)  # Ensure bid is lower than ask

            return ('buy', bid_price, order_size)
        elif direction == 'sell':
            ask_price = price
            bid_price = self.round_to_tick(recent_prices[-1] * (1 - 0.005))  # Slightly below the current price

            if ask_price <= bid_price:
                ask_price = self.round_to_tick(bid_price + 0.05)  # Ensure ask is higher than bid

            return ('sell', ask_price, order_size)

    def calculate_pnl(self, order_type, price, size):
        """Calculate PnL based on the current position and the trade being executed."""
        if self.current_position is None:
            trade_pnl = 0
        elif order_type == "sell":  # PnL only at closing a trade
            trade_pnl = (price - self.current_position['price']) * size
        elif order_type == "buy":  # PnL only at closing a trade
            trade_pnl = (self.current_position['price'] - price) * size

        return trade_pnl

    def order_executed(self, trade_price, trade_size, trade_type, current_time):
        if trade_type == "buy":
            if self.current_position:
                # Closing Existing SELL Position by Buying
```

```
                self.inventory += trade_size
                self.cash -= trade_price * trade_size
                trade_pnl = self.calculate_pnl(trade_type, trade_price, trade_size)
                self.pnl += trade_pnl  # Update cumulative PnL
                self.record_metrics(current_time, trade_price, trade_size, trade_type, trade_pnl, "close")
                self.order_history.append((trade_price, trade_size))
                self.current_position = None  # Reset current position after closing
            else:
                # Opening New BUY Position
                self.hold_time = random.randint(30, 240)  # Set a random holding period
                self.last_trade_time = current_time  # Set the time when the position was opened
                self.inventory += trade_size
                self.cash -= trade_price * trade_size
                trade_pnl = self.calculate_pnl(trade_type, trade_price, trade_size)
                self.pnl += trade_pnl  # Update cumulative PnL
                self.record_metrics(current_time, trade_price, trade_size, trade_type, trade_pnl, "open")
                self.order_history.append((trade_price, trade_size))
                self.current_position = {"type": trade_type, "price": trade_price, "size": abs(trade_size)}

        elif trade_type == "sell":
            if self.current_position:
                # Closing Existing BUY Position by Selling
                self.inventory -= trade_size
                self.cash += trade_price * trade_size
                trade_pnl = self.calculate_pnl(trade_type, trade_price, trade_size)
                self.pnl += trade_pnl  # Update cumulative PnL
                self.record_metrics(current_time, trade_price, trade_size, trade_type, trade_pnl, "close")
                self.order_history.append((trade_price, trade_size))
                self.current_position = None  # Reset current position after closing
            else:
                # Opening New SELL Position
                self.hold_time = random.randint(30, 240)  # Set a random holding period
                self.last_trade_time = current_time  # Set the time when the position was opened
                self.inventory -= trade_size
                self.cash += trade_price * trade_size
                trade_pnl = self.calculate_pnl(trade_type, trade_price, trade_size)
                self.pnl += trade_pnl  # Update cumulative PnL
                self.record_metrics(current_time, trade_price, trade_size, trade_type, trade_pnl, "open")
                self.order_history.append((trade_price, trade_size))
                self.current_position = {"type": trade_type, "price": trade_price, "size": abs(trade_size)}

    def record_metrics(self, timestamp, price, size, order_type, trade_pnl, position):
        """Record relevant metrics in the DataFrame."""
        new_entry = pd.DataFrame({
            "momentum_trader": self.momentum_tag,
            "aggressiveness": self.aggressiveness,
```

```
                "timestamp": [timestamp],
                "cash": [self.cash],
                "inventory": [self.inventory],
                "pnl": [trade_pnl],
                "order_type": [order_type],
                "order_price": [price],
                "order_size": [size],
                "order_dollar_value": [price * size],
                "position": position
            })
            self.metrics_df = pd.concat([self.metrics_df, new_entry], ignore_index=True)

    def record_metrics_for_liquidation(self):
        """Update the last entry in the DataFrame to reflect the negative cash balance."""
        if not self.metrics_df.empty:
            self.metrics_df.loc[self.metrics_df.index[-1], "cash"] = self.cash

    def export_metrics(self):
        """Export the metrics DataFrame to a CSV file."""
        file_name = f"uninformed_investor_{self.id}_metrics.csv"
        self.metrics_df.to_csv(f"data/uninformed_investors/{file_name}", index=False)
        print(f"Metrics for Uninformed Investor {self.id} exported to {file_name}.")

class MarketMaker(Agent):
    def __init__(self, id, initial_cash, initial_inventory, aggressiveness=0.1, risk_aversion=0.05):
        super().__init__(id, initial_cash, initial_inventory, aggressiveness)
        self.risk_aversion = risk_aversion
        self.spread_history = []  # Track bid-ask spread over time
        self.volume_history = []  # Track quoted volumes over time
        self.inventory_history = []  # Track inventory over time
        self.pnl_history = []  # Track combined PnL over time
        self.realized_pnl_history = []  # Track realized PnL separately
        self.unrealized_pnl_history = []  # Track unrealized PnL separately
        self.realized_pnl = 0  # Track realized PnL separately
        self.unrealized_pnl = 0  # Track unrealized PnL separately
        self.distance_to_best_bid = []
        self.distance_to_best_ask = []
        self.fill_rate_history = []
        self.executed_orders = []  # Track executed orders separately
        self.time_history = []  # Track timestamps for analysis
        self.is_liquidated = False

    def round_to_tick(self, price):
        """Round price to the nearest 0.05 increment."""
        return round(price * 20) / 20.0

    def calculate_optimal_prices(self, current_price, inventory, sigma, T, kappa, best_bid, best_ask):
        """
        Calculate optimal bid and ask prices using the Avellaneda-Stoikov model.

        :param current_price: The current mid-price of the asset.
        :param inventory: The current inventory level.
        :param sigma: The volatility of the asset.
```

```python
        :param T: The time horizon for the market maker.
        :param kappa: The market impact parameter.
        :return: The optimal bid and ask prices.
        """
        # Calculate the reservation price
        reservation_price = current_price - (inventory * self.risk_aversion * sigma**2 * T)

        # Calculate the optimal bid and ask prices using the Avellaneda-Stoikov model
        bid_price = self.round_to_tick(reservation_price - (1 / self.risk_aversion) * np.log(1 + self.risk_aversion / kappa))
        ask_price = self.round_to_tick(reservation_price + (1 / self.risk_aversion) * np.log(1 + self.risk_aversion / kappa))

        # Ensure the bid is lower than the ask
        if bid_price >= ask_price:
            ask_price = bid_price + 0.05

        self.order_history.append((bid_price, ask_price))
        self.spread_history.append(ask_price - bid_price)

        # Track distance from best bid/ask
        self.distance_to_best_bid.append(bid_price - best_bid)
        self.distance_to_best_ask.append(ask_price - best_ask)

        return bid_price, ask_price


    def calculate_optimal_volume(self, current_price, order_book, inventory, best_bid_volume, best_ask_volume):
        # Factor 1: Adjust based on inventory
        inventory_factor = max(0.5, 1 - 0.5 * self.risk_aversion * abs(inventory))  # Reduce impact, increase volume

        # Extract all buy and sell volumes
        buy_volumes = [order["size"] for orders_at_price in order_book.buy_orders.values() for order in orders_at_price]
        sell_volumes = [order["size"] for orders_at_price in order_book.sell_orders.values() for order in orders_at_price]

        # Combine buy and sell volumes to get the overall depth
        all_volumes = buy_volumes + sell_volumes

        if len(all_volumes) > 0:
            depth_factor = min(2, np.mean(all_volumes) / 5000)  # Increase volume with higher market depth
        else:
            depth_factor = 0.5  # Base depth factor

        # Factor 3: Adjust based on wealth and risk aversion
        wealth_factor = max(0.5, self.cash / (self.cash + inventory * current_price)) * (1 - self.risk_aversion)

        # Final optimal volume calculation
        optimal_volume = inventory_factor * depth_factor * wealth_factor * 1e3

        # Make volume close to the best bid/ask volumes
        optimal_volume = min(optimal_volume, best_bid_volume * 0.8, best_ask_volume * 0.8)

```

```
        return max(100, round(optimal_volume, 2))  # Ensure the volume is at least 100

    def place_order(self, current_price, order_book, inventory, sigma, T, kappa):
        best_bid = max(order_book.buy_orders.keys()) if order_book.buy_orders else current_price
        best_ask = min(order_book.sell_orders.keys()) if order_book.sell_orders else current_price

        bid_price, ask_price = self.calculate_optimal_prices(current_price, inventory, sigma, T, kappa, best_bid, best_ask)
        bid_volume = self.calculate_optimal_volume(bid_price, order_book, inventory, sum([order["size"] for order in order_book.buy_orders.get(best_bid, [])]), sum([order["size"] for order in order_book.sell_orders.get(best_ask, [])]))
        ask_volume = self.calculate_optimal_volume(ask_price, order_book, inventory, sum([order["size"] for order in order_book.buy_orders.get(best_bid, [])]), sum([order["size"] for order in order_book.sell_orders.get(best_ask, [])]))

        # Store the order in the order history
        self.order_history.append(("buy", bid_price, bid_volume))
        self.order_history.append(("sell", ask_price, ask_volume))
        self.volume_history.append((bid_volume, ask_volume))
        self.inventory_history.append(inventory)

        return bid_price, bid_volume, ask_price, ask_volume

    def order_executed(self, trade_price, trade_size, trade_type, current_time):
        if trade_type == "buy":
            self.inventory += trade_size
            self.cash -= trade_price * trade_size
            # Realized PnL for closing short position
            if self.inventory < 0:
                self.realized_pnl += (self.last_trade_price - trade_price) * trade_size
        elif trade_type == "sell":
            self.inventory -= trade_size
            self.cash += trade_price * trade_size
            # Realized PnL for closing long position
            if self.inventory > 0:
                self.realized_pnl += (trade_price - self.last_trade_price) * trade_size

        # Record the executed trade in executed_orders
        self.executed_orders.append((trade_type, trade_price, trade_size, current_time))

        # Update the last trade price to mark-to-market the current inventory
        self.last_trade_price = trade_price

        # Update PnL after each trade
        self.unrealized_pnl = self.inventory * trade_price  # Mark-to-market value of inventory
        total_pnl = self.realized_pnl + self.unrealized_pnl
        self.pnl_history.append(total_pnl)

        if self.cash < 0:
            self.is_liquidated = True  # Liquidate if cash is negative
            print("MARKET MAKER HAS GONE BANKRUPT")
```

```python

        # Update separate PnL histories
        self.update_pnl_histories()

    def update_pnl_histories(self):
        """Update the separate histories for realized and unrealized PnL."""
        self.realized_pnl_history.append(self.realized_pnl)
        self.unrealized_pnl_history.append(self.unrealized_pnl)

    def calculate_fill_rate(self):
        # Total number of orders placed (bid + ask)
        total_orders = len(self.order_history)
        # Number of orders executed (filled)
        filled_orders = len(self.executed_orders)
        # Calculate fill rate
        fill_rate = filled_orders / total_orders if total_orders > 0 else 0
        self.fill_rate_history.append(fill_rate)

    def analyze_fill_rate_vs_time(self):
        """Plot Fill Rate over time."""
        # self.calculate_fill_rate()
        plt.figure(figsize=(10, 6))
        plt.plot(range(len(self.fill_rate_history)), self.fill_rate_history, color='orange')
        plt.xlabel('Time')
        plt.ylabel('Fill Rate')
        plt.title('Fill Rate Over Time')
        plt.grid(True)
        plt.savefig("data/fill_rate.png", dpi=300)

    def analyze_distance_to_best_vs_time(self):
        """Plot distance to best bid/ask over time."""
        plt.figure(figsize=(10, 6))
        plt.plot(range(len(self.distance_to_best_bid)), self.distance_to_best_bid, color='blue', label='Distance to Best Bid')
        plt.plot(range(len(self.distance_to_best_ask)), self.distance_to_best_ask, color='red', label='Distance to Best Ask')
        plt.xlabel('Time')
        plt.ylabel('Distance to Best Bid/Ask')
        plt.title('Distance to Best Bid/Ask Over Time')
        plt.legend()
        plt.grid(True)
        plt.savefig("data/distance_to_best.png", dpi=300)

    def analyze_spread_vs_inventory(self):
        """Plot the bid-ask spread as a function of inventory."""
        plt.figure(figsize=(10, 6))
        plt.plot(self.inventory_history, self.spread_history, 'o-', color='purple')
        plt.xlabel('Inventory')
        plt.ylabel('Bid-Ask Spread')
        plt.title('Bid-Ask Spread vs Inventory')
        plt.grid(True)
        plt.savefig("data/spread_vs_inventory.png", dpi=300)

    def analyze_volume_vs_inventory(self):
        """Plot the quoted volumes as a function of inventory."""
        bid_volumes, ask_volumes = zip(*self.volume_history)
        plt.figure(figsize=(10, 6))
        plt.plot(self.inventory_history, bid_volumes, 'o-', color='blue', label='Bid Volume')
```

```
        plt.plot(self.inventory_history, ask_volumes, 'o-', color='orange', label='Ask Volume')
        plt.xlabel('Inventory')
        plt.ylabel('Volume')
        plt.title('Quoted Volumes vs Inventory')
        plt.legend()
        plt.grid(True)
        plt.savefig("data/volume_vs_inventory.png", dpi=300)

    def analyze_pnl_vs_inventory(self):
        """Plot PnL and Inventory as a function of time on two subplots."""
        fig, ax1 = plt.subplots(2, 1, figsize=(10, 12))

        # Subplot 1: PnL vs Inventory
        ax1[0].plot(range(len(self.pnl_history)), self.pnl_history, 'o-', color='green')
        ax1[0].set_xlabel('Inventory')
        ax1[0].set_ylabel('PnL')
        ax1[0].set_title('PnL vs Inventory')
        ax1[0].grid(True)

        # Subplot 2: Inventory over time
        ax1[1].plot(range(len(self.inventory_history)), self.inventory_history, 'o-', color='blue')
        ax1[1].set_xlabel('Time')
        ax1[1].set_ylabel('Inventory')
        ax1[1].set_title('Inventory Over Time')
        ax1[1].grid(True)

        plt.tight_layout()  # Adjust subplots to fit in the figure area.
        plt.savefig("data/pnl_vs_inventory.png", dpi=300)

    def analyze_pnl_vs_time(self):
        """Plot PnL over time."""
        plt.figure(figsize=(10, 6))
        plt.plot(range(len(self.pnl_history)), self.pnl_history, color='red')
        plt.xlabel('Time')
        plt.ylabel('PnL')
        plt.title('PnL Over Time')
        plt.grid(True)
        plt.savefig("data/PnL.png", dpi=300)

    def analyze_inventory_vs_time(self):
        """Plot inventory over time."""
        plt.figure(figsize=(10, 6))
        plt.plot(range(len(self.inventory_history)), self.inventory_history, color='blue')
        plt.xlabel('Time')
        plt.ylabel('Inventory')
        plt.title('Inventory Over Time')
        plt.grid(True)
        plt.savefig("data/inventory_over_time.png", dpi=300)

    def analyze_realized_pnl(self):
        """Plot Realized PnL over time."""
        plt.figure(figsize=(10, 6))
        plt.plot(range(len(self.realized_pnl_history)), self.realized_pnl_history, color='green')
        plt.xlabel('Time')
        plt.ylabel('Realized PnL')
```

```python
        plt.title('Realized PnL Over Time')
        plt.grid(True)
        plt.savefig("data/realized_pnl.png", dpi=300)

    def analyze_unrealized_pnl(self):
        """Plot Unrealized PnL over time."""
        plt.figure(figsize=(10, 6))
        plt.plot(range(len(self.unrealized_pnl_history)), self.unrealized_pnl_history, color='orange')
        plt.xlabel('Time')
        plt.ylabel('Unrealized PnL')
        plt.title('Unrealized PnL Over Time')
        plt.grid(True)
        plt.savefig("data/unrealized_pnl.png", dpi=300)


    def run_all_analyses(self):
        """Run all analyses to visualize the Market Maker's performance."""
        self.analyze_spread_vs_inventory()
        self.analyze_volume_vs_inventory()
        self.analyze_pnl_vs_time()
        self.analyze_inventory_vs_time()
        self.analyze_fill_rate_vs_time()
        self.analyze_distance_to_best_vs_time()
        self.analyze_realized_pnl()
        self.analyze_unrealized_pnl()

class OrderBook:
    def __init__(self):
        self.buy_orders = collections.defaultdict(list)
        self.sell_orders = collections.defaultdict(list)
        self.trade_ledger = []

    def add_order(self, agent_id, order_type, price, size, timestamp):
        order = {"agent_id": agent_id, "price": price, "size": size, "timestamp": timestamp}

        if order_type == "buy":
            self.buy_orders[price].append(order)
        elif order_type == "sell":
            self.sell_orders[price].append(order)

    def execute_trades(self, agents, current_price, timestamp):
        executed_trades = []

        # Sort buy and sell orders by price (highest buy first, lowest sell first)
        buy_prices = sorted(self.buy_orders.keys(), reverse=True)
        sell_prices = sorted(self.sell_orders.keys())

        # Execute trades where buy price >= sell price
        for buy_price in buy_prices:
            if buy_price < current_price:
                continue
            for sell_price in sell_prices:
                if sell_price > current_price or buy_price < sell_price:
                    continue

                buy_orders = self.buy_orders[buy_price]
                sell_orders = self.sell_orders[sell_price]

```

```python
                # Sort by timestamp, then by volume for priority execution
                buy_orders.sort(key=lambda x: (x["timestamp"], -x["size"]))
                sell_orders.sort(key=lambda x: (x["timestamp"], -x["size"]))

                while buy_orders and sell_orders:
                    buy_order = buy_orders[0]
                    sell_order = sell_orders[0]

                    # Determine the trade size
                    trade_size = min(buy_order["size"], sell_order["size"])

                    # Execute trade
                    self.trade_ledger.append({
                        "timestamp": timestamp,
                        "buyer_id": buy_order["agent_id"],
                        "seller_id": sell_order["agent_id"],
                        "price": sell_price,
                        "size": trade_size
                    })
                    executed_trades.append(self.trade_ledger[-1])

                    # Update order sizes
                    buy_order["size"] -= trade_size
                    sell_order["size"] -= trade_size

                    # Order Executed - update each agent's ledger and metrics
                    if buy_order["agent_id"] == market_maker.id:
                        market_maker.order_executed(sell_price, trade_size, "buy", timestamp)
                    else:
                        agents[buy_order["agent_id"] - 1].order_executed(sell_price, trade_size, "buy", timestamp)

                    if sell_order["agent_id"] == market_maker.id:
                        market_maker.order_executed(sell_price, trade_size, "sell", timestamp)
                    else:
                        agents[sell_order["agent_id"] - 1].order_executed(sell_price, trade_size, "sell", timestamp)


                    # Remove orders if completely filled
                    if buy_order["size"] == 0:
                        buy_orders.pop(0)
                    if sell_order["size"] == 0:
                        sell_orders.pop(0)

                # Remove price level if no orders left
                if not buy_orders:
                    del self.buy_orders[buy_price]
                if not sell_orders:
                    del self.sell_orders[sell_price]

        return executed_trades

    def get_order_book_snapshot(self):
        """Return a snapshot of the current order book."""
```

```python
        buy_snapshot = {price: sum(order["size"] for order in orders) for price, orders in self.buy_orders.items()}
        sell_snapshot = {price: sum(order["size"] for order in orders) for price, orders in self.sell_orders.items()}
        return buy_snapshot, sell_snapshot

    def plot_order_book(self, market_maker_bid=None, market_maker_bid_volume=None,
                        market_maker_ask=None, market_maker_ask_volume=None, title="Order Book Snapshot"):
        """Plot the order book as a depth chart."""
        buy_snapshot, sell_snapshot = self.get_order_book_snapshot()

        # Sort the prices
        buy_prices = sorted(buy_snapshot.keys(), reverse=True)
        sell_prices = sorted(sell_snapshot.keys())

        # Cumulative volumes
        buy_volumes = np.cumsum([buy_snapshot[price] for price in buy_prices])
        sell_volumes = np.cumsum([sell_snapshot[price] for price in sell_prices])

        plt.figure(figsize=(10, 6))

        # Plot Bids (Buy orders)
        plt.step(buy_prices, buy_volumes, where='mid', color='green', alpha=0.7, label='Bids')

        # Plot Asks (Sell orders)
        plt.step(sell_prices, sell_volumes, where='mid', color='red', alpha=0.7, label='Asks')

        # Plot Market Maker's Bid
        if market_maker_bid is not None and market_maker_bid_volume is not None:
            plt.axvline(x=market_maker_bid, color='blue', linestyle='--', label='Market Maker Bid')
            plt.scatter(market_maker_bid, market_maker_bid_volume, color='blue', label='MM Bid Vol')

        # Plot Market Maker's Ask
        if market_maker_ask is not None and market_maker_ask_volume is not None:
            plt.axvline(x=market_maker_ask, color='orange', linestyle='--', label='Market Maker Ask')
            plt.scatter(market_maker_ask, market_maker_ask_volume, color='orange', label='MM Ask Vol')

        plt.xlabel('Price')
        plt.ylabel('Cumulative Volume')
        plt.title(title)
        plt.legend()
        plt.grid(True)
        plt.savefig(f"data/{title.replace(' ', '_').lower()}.png", dpi=300)
        plt.show()

# Analysis Class
class Analysis:
    def __init__(self):
        self.data = []

```

```python
    def collect_data(self, agent, current_price, is_market_maker=False)
:
        self.data.append({
            "agent_id": agent.id,
            "cash": agent.cash,
            "inventory": agent.inventory,
            "current_price": current_price,
            "is_market_maker": is_market_maker
        })

    def create_dataframe(self):
        df = pd.DataFrame(self.data)
        return df

    def analyze_profit(self, df):
        initial_cash = df.groupby('agent_id')['cash'].first()
        df['PnL'] = df['cash'] + df['inventory'] * df['current_price']
 - initial_cash[df['agent_id']].values
        return df

    def plot_market_maker_pnl(self, df):
        market_maker_data = df[df['is_market_maker']]

        # Create a figure and two subplots
        fig, axs = plt.subplots(1, 1, figsize=(10, 6))

        # Plot PnL in the first subplot
        axs.plot(market_maker_data.index, market_maker_data['PnL'],
 label='Market Maker PnL', color='red')
        axs.set_title('Market Maker PnL Over Time')
        axs.set_xlabel('Time')
        axs.set_ylabel('PnL')
        axs.legend()
        axs.grid(True)

        # Adjust layout to prevent overlap
        plt.tight_layout()
        plt.savefig("data/market_maker_pnl.png", dpi=300)

    def plot_market_maker_inventory(self, df):
        market_maker_data = df[df["is_market_maker"]]
        # Create a figure and two subplots
        fig, axs = plt.subplots(1, 1, figsize=(10, 6))
        # Plot Inventory in the second subplot
        axs.plot(market_maker_data.index, market_maker_data['inventory'
], label='Market Maker Inventory', color='blue')
        axs.set_title('Market Maker Inventory Over Time')
        axs.set_xlabel('Time')
        axs.set_ylabel('Inventory')
        axs.legend()
        axs.grid(True)
        # Adjust layout to prevent overlap
        plt.tight_layout()
        plt.savefig("data/market_maker_inventory.png", dpi=300)


def pause_and_resume():
    while True:
        user_input = input("Press 1 to continue or 2 to break: ")  #
 Prompt inside the loop
        if user_input == '1':  # Check against the string '1'
            print("Continuing...")
            break  # This will exit the loop and continue the program
```

```
        elif user_input == '2':  # Check against the string '2'
            print("Exiting program...")
            exit()  # Exit the entire program
        else:
            print("Invalid input. Please press 1 to continue or 2 to break.")

# Simulation Setup
if __name__ == "__main__":
    # Simulate Stock Prices
    seed = 50
    simulator = PriceSimulator(days=3, initial_price=100, mu=0.0001, sigma=0.25, dt=1, seed=seed)
    prices = simulator.simulate_brownian_motion_prices()
    price_df = simulator.create_dataframe(prices)
    simulator.plot_prices()

    # Market Maker Parameters
    T = 1 # Time horizon for the market maker (one trading day divided by the number of minutes per day)
    kappa = 0.5  # Market impact parameter (example value)

    # Agents Setup
    np.random.seed(seed)
    MARKET_MAKER = 1
    UNINFORMED_INVESTORS = 1000 - MARKET_MAKER

    aggressiveness_values = np.random.uniform(0, 0.4, UNINFORMED_INVESTORS)

    uninformed_investors = [UninformedInvestorAgent(id=i+1, initial_cash=100_000, initial_inventory=0, aggressiveness=aggressiveness_values[i]) for i in range(UNINFORMED_INVESTORS)]
    market_maker = MarketMaker(id=UNINFORMED_INVESTORS+1, initial_cash=10_000_000, initial_inventory=0, risk_aversion=0.5)

    agents = uninformed_investors

    # Initialize OrderBook and Analysis
    order_book = OrderBook()
    analysis = Analysis()

    # Run the Simulation
    window_size = 480  # Last 1 days of trading

    for t, price in enumerate(prices):
        if t < window_size:
            continue  # Wait until we have enough data for the uninformed investors

        # Future price is simply the next price in the simulation
        future_price = prices[t + 120] if t + 120 < len(prices) else price
        recent_prices = prices[t-window_size:t]  # Last 5 days

        # Calculate sigma from recent_prices
        log_returns = np.log(np.array(recent_prices[1:]) / np.array(recent_prices[:-1]))
        sigma = np.std(log_returns)

        print(f"Time: {t}, Current Price:{recent_prices[-1]}, Future Price: {future_price}")

```

```python
    try:
        print(f"Market Maker PnL: {market_maker.pnl_history[-1]}")
    except:
        pass

    for agent in agents:
        order = None  # Initialize order to None for each agent
        if isinstance(agent, UninformedInvestorAgent):
            order = agent.place_order(recent_prices, current_time=t)
        # Add valid orders to the order book
        if order:
            order_book.add_order(agent.id, order[0], order[1], size=order[2], timestamp=t)

    # Market Maker places orders based on its strategy
    bid, bid_volume, ask, ask_volume = market_maker.place_order(current_price=price, order_book=order_book, inventory=market_maker.inventory, sigma=sigma, T=T, kappa=kappa)
    # bid, bid_volume, ask, ask_volume = market_maker.place_order(current_price=price, order_book=order_book, inventory=market_maker.inventory)
    order_book.add_order(market_maker.id, "buy", bid, size=bid_volume, timestamp=t)
    order_book.add_order(market_maker.id, "sell", ask, size=ask_volume, timestamp=t)
    # order_book.plot_order_book(market_maker_bid=bid,
    #                            market_maker_bid_volume=bid_volume,
    #                            market_maker_ask=ask,
    #                            market_maker_ask_volume=ask_volume)
    # print(f"Market Maker Bid: {bid}, Bid Volume: {bid_volume}, Ask: {ask}, Ask Volume: {ask_volume}")
    # pause_and_resume()
    # Execute orders
    executed_orders = order_book.execute_trades(agents=agents, current_price=price, timestamp=t)
    # Calculate fill rate after trades are executed
    market_maker.calculate_fill_rate()
    if market_maker.is_liquidated:
        break
    # Collect data for analysis
    for agent in agents:
        analysis.collect_data(agent, price)
    analysis.collect_data(market_maker, price, is_market_maker=True)

# Analyze and Plot Results
df = analysis.create_dataframe()
df = analysis.analyze_profit(df)
df.to_csv("data/analysis.csv", index=False)
market_maker.run_all_analyses()
analysis.plot_market_maker_pnl(df)
analysis.plot_market_maker_inventory(df)

print("Simulation Complete")
```

LISTING A.1: Market Maker Simulation