ISTANBUL TECHNICAL UNIVERSITY
Faculty of Computer Science and Informatics

STOCK MARKET SIMULATION APPLICATION

INTERNSHIP PROGRAM REPORT

UMUTCAN DOĞAN
150200712

SUMMER / 2023

Istanbul Technical University

Faculty of Computer Science and Informatics

INTERNSHIP REPORT

Academic Year:	2023/2024
Internship Term:	☒ Summer ☐ Spring ☐ Fall

Student Information

Name Surname:	UMUTCAN DOĞAN
Student ID:	150200712
Department:	Computer Engineering
Program:	100% English
E-Mail:	doganumu20@itu.edu.tr
Mobile Phone:	+90 (506) 132 4373
Pursuing a Double Major?	☐ Yes ☒ No
In the Graduation Term?	☒ Yes ☐ No
Taking a class at Summer School?	☐ Yes ☒ No

Institution Information

Company Name:	Türk Ekonomi Bankası A.Ş.
Department:	Software
Web Address:	https://www.teb.com.tr/
PostalAddress:	Saray, Sokullu Cd 7A C Blok No: 7/A, 34768 Ümraniye/İstanbul

Authorized Person Information

Department:	Software
Title:	Department Leader
Name Surname:	Ferhat Rızaoğlu
Corporate E-Mail:	ferhat.rizaoglu@teb.com.tr
Corporate Phone:	-

Internship Work Information

Internship Location:	☒ Turkey ☐ Abroad
Internship Start Date:	07.08.2023
Internship End Date:	04.09.2023
Number of Days Worked:	20
During your internship, did you have insurance?	☒ Yes, I was insured by İTÜ. ☐ Yes, I was insured by institution. ☐ No, I did my internship abroad. ☐ No.

Table of Contents

1 INFORMATION ABOUT THE INSTITUTION

The Türk Ekonomi Bankası (TEB) is a private deposit bank headquartered in Istanbul, Turkey[1]. It was originally established in 1927 under the name Kocaeli Halk Bankası T.A.Ş. and operated under this name until 1982. Starting from that date, the bank began operating nationwide, and its headquarters were relocated to Istanbul. Its name was changed to Türk Ekonomi Bankası A.Ş. (TEB). In 2005, the bank entered into a strategic partnership agreement with BNP Paribas, one of the leading banks in the Eurozone and the world.

Türk Ekonomi Bankası is the third oldest bank in Turkey, following Ziraat Bankası (1863) and Türkiye İş Bankası (1924) among the active banks in the country.

TEB has a total of 471 branches in 71 cities (4 of which are located abroad) and 1,704 ATMs, offering corporate, SME, Treasury and Capital Markets, personal, and private banking services to its customers. Additionally, through its subsidiaries and group companies, the bank provides financial services and products in areas such as investment, leasing, factoring, and portfolio management.

TEB is involved in a wide range of projects, including innovation, sports, financial literacy, entrepreneurship, projects for SMEs, and initiatives supporting female entrepreneurs. TEB's Entrepreneurship Banking unit works to reach entrepreneurs across the country, especially tech startups, and help bring their innovative business ideas to the economy. The bank also continues to engage in financial projects in collaboration with fintech companies in the financial sector.

2 INTRODUCTION

My project involves the development of a stock market web application. The website will present a user-friendly interface where stocks are organized in columns. These columns will display real-time updates, with stock prices changing dynamically. The design incorporates a color-coded system, with stocks turning green when their prices rise and red when they decline, aligning with the conventional visual representation used in stock markets.

3 DESCRIPTION AND ANALYSIS OF THE INTERNSHIP PROJECT

The project involves the implementation of a dynamic stock market simulation with a database infrastructure. Autonomous agents will continuously access the database, executing random buying or selling transactions with initial monetary resources. The database will be consistently updated to reflect these agent interactions. Additionally, the web application will provide a user login feature, allowing users to manage their accounts. Users can check their available funds, engage in stock transactions, and monitor their stock portfolio and transaction history.

Each agent will commence with an initial balance, determined randomly from a Gaussian distribution centered around the arithmetic mean of total balances of real-life stock market users. Subsequently, agents will randomly select a company to invest in. They will then invest their entire balance in the chosen company's stocks, considering the current stock price. Additionally, agents will have an upper limit for investment, randomly chosen from a Gaussian distribution centered at the prevailing stock price of the selected company.

The program consists of several classes, including the 'Agent' class with attributes 'stocks' initialized to 0 and 'balance' randomly chosen from a normal distribution centered at 10. Another class, 'Company,' is defined with attributes 'stocks' set to 1000 and 'curr price' initialized to 1. The 'Trader' class is equipped with two PriorityQueues – one for buying and one for selling. Its 'trade' function is invoked whenever either of the queues is updated, ensuring dynamic market interactions. The 'curr price' is updated each time the selling queue changes, where the minimum value in the selling queue becomes the new current price.

In the main program, the company initiates by selling all its stocks, triggering an update to the selling queue via the 'sell' function (e.g., sell(1.0, 1000)). Subsequently, 100 agents are created. Upon creation, each agent places a buying offer in the buying queue using the 'buy' function. The 'buy' function determines the maximum amount an agent can buy with its balance and randomly selects a price from a normal distribution centered around the current price. If an agent successfully makes a purchase, it then places a corresponding selling offer in the selling queue. The 'sell' function mirrors the 'buy' mechanism, selling all stocks at a randomly chosen price. Additionally, the program displays the current stock price on the terminal whenever the 'curr price' changes.

To do that first I needed to make some project planning and design. I created a class diagram.

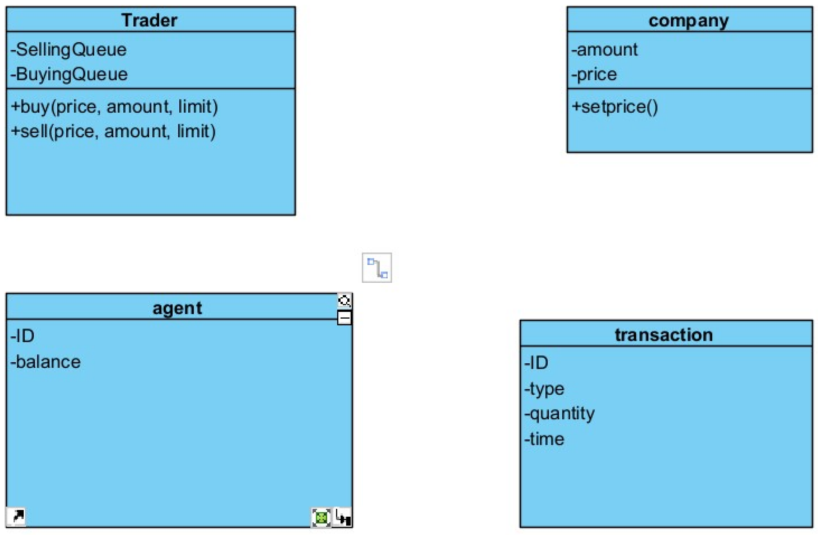


Figure 1: Class diagram of the project

Figure 1 is just a draft, but it gives some intuition for my project. I created a github directory and committed the first working version of the code. There is only one company for now, and the code can correctly display the current stock price of the company, and it seems that it randomly fluctuates around new current prices.

3.1 Explanation of the First Version

3.1.1 RandomNumberGenerator Class (RandomNumberGenerator)

This class generates random numbers using the Box-Muller transform to approximate a Gaussian distribution. This is handy for creating random prices for buying and selling offers.

3.1.2 Agent Class (Agent)

An agent is like a person or entity participating in the stock market. Each agent has a certain amount of money called ”balance” and some stocks. With the buy() method, an agent can use their balance to buy stocks at a given price. Conversely, the sell() method allows an agent to sell all their stocks at a certain price.

3.1.3 Trader Class (Trader)

The Trader class is the part of the code that handles the trading process. It maintains the buying and selling queues where trade offers are stored. The main action is in the trade() method, which matches buying and selling offers based on their prices.

- In trade(), the loop runs as long as there are matching offers, where the highest buying price is higher than or equal to the lowest selling price.
- If the buyer wants to buy more stocks than the seller is offering, they trade. The buyer buys the amount the seller is offering, and the seller's stocks become zero.
- If the seller offers more stocks than the buyer wants to buy, a trade occurs, and the buyer's desired buying amount is reduced.
- The addBuyingOffer() and addSellingOffer() methods are used to add new offers to the respective queues. After an offer is added, trade() is called immediately to see if trades can occur.

3.1.4 TradeOffer Class (TradeOffer)

A TradeOffer represents an offer to trade stocks between two agents at a particular price and amount. By implementing the Comparable interface, offers can be sorted based on their price.

3.1.5 Company Class (Company)

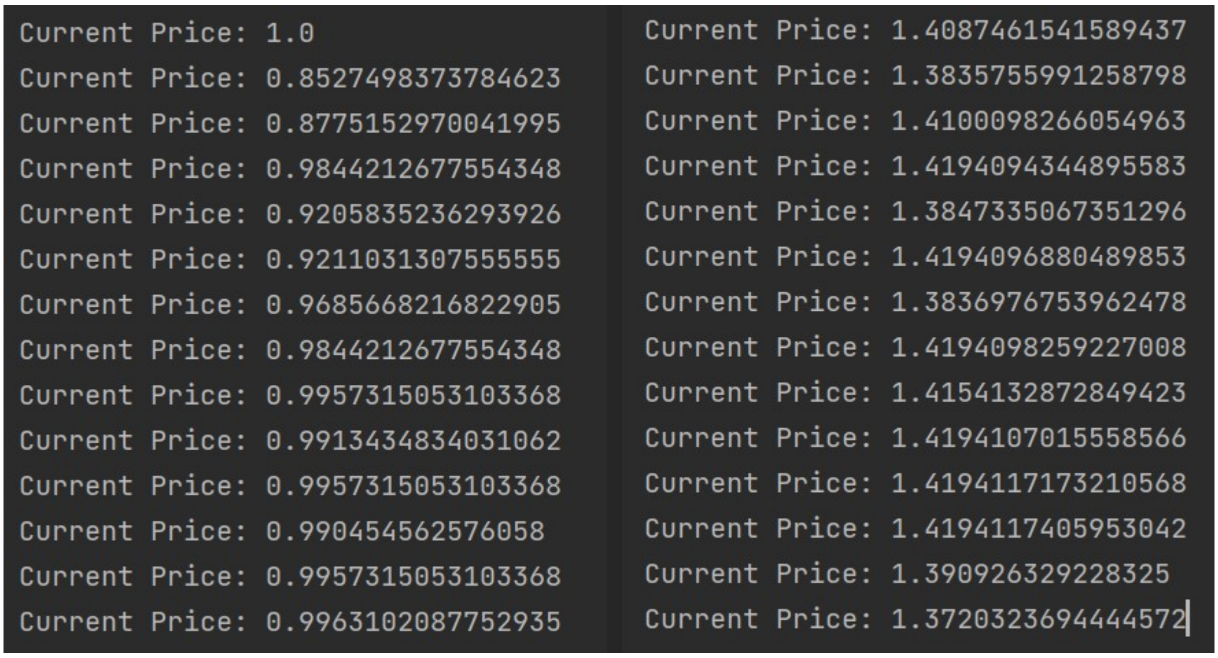
The Company class represents a company that sells stocks. The sell() method is meant for the company to sell stocks to agents through the trader. However, in this version of the code, the company interaction is not fully developed.

3.1.6 StockMarketSimulation Class (StockMarketSimulation)

This class is the main part of the simulation. It sets up the different components and simulates the stock market.

- It creates a Trader instance and a Company instance.
- The company agent is set up with zero balance, and the company uses the sell() method to sell 1000 stocks to the trader.
- In an infinite loop, the simulation generates new agents with random balances, which represent potential buyers in the market. It initiates buying offers for them.
- If an agent's balance allows them to buy some stocks, a corresponding selling offer is added.
- After adding each offer, the trade() method is called to see if any trades can be executed.
- The current price is printed whenever it changes.

Here is a screenshot of the output on terminal:



```
Current Price: 1.0
Current Price: 0.8527498373784623
Current Price: 0.8775152970041995
Current Price: 0.9844212677554348
Current Price: 0.9205835236293926
Current Price: 0.9211031307555555
Current Price: 0.9685668216822905
Current Price: 0.9844212677554348
Current Price: 0.9957315053103368
Current Price: 0.9913434834031062
Current Price: 0.9957315053103368
Current Price: 0.990454562576058
Current Price: 0.9957315053103368
Current Price: 0.9963102087752935
```

```
Current Price: 1.4087461541589437
Current Price: 1.3835755991258798
Current Price: 1.4100098266054963
Current Price: 1.4194094344895583
Current Price: 1.3847335067351296
Current Price: 1.4194096880489853
Current Price: 1.3836976753962478
Current Price: 1.4194098259227008
Current Price: 1.4154132872849423
Current Price: 1.4194107015558566
Current Price: 1.4194117173210568
Current Price: 1.4194117405953042
Current Price: 1.390926329228325
Current Price: 1.3720323694444572
```

Figure 2: Screenshot of the program

As it can be seen from Figure 2, the more we run the program (screenshot on the right was took after some time), more it gets randomly far from the starting price, which is 1.

3.2 Explanation of the Second Version

3.2.1 Trader Class (Trader)

The Trader class handles the trading process and keeps track of trade offers. It has been modified and expanded with additional features.

- A new property, compStock, is added to store the total stocks of the company. This value is set through the constructor.
- In the trade() method, a list named agents is introduced to keep track of agents involved in trades.
- There's a condition that checks if the highest buying price is lower than the lowest selling price. If true, a new agent is created. This agent has a balance based on a random number, and a buying offer is initiated for them. This helps simulate new potential buyers entering the market.
- The main part of the trade() method has been adjusted. It processes trades by comparing the buying and selling prices. If a buyer's price is higher than the current selling price, the method starts processing trades between buyers and sellers.
- When matching trades occur, the buyer buys from the seller, and the stocks and balances are updated accordingly. The total amount of stocks traded and the agents involved are tracked.
- After successful trades, new selling offers are added for the remaining stocks. Additionally, new buying offers are generated for agents that still have a balance. This simulates continuous market activity.

3.2.2 StockMarketSimulation Class (StockMarketSimulation)

The StockMarketSimulation class orchestrates the simulation of the stock market and includes the main method.

- The Company class is instantiated with 200 stocks and an initial price of 1.
- The Trader class is instantiated with the company's stock information.
- In the simulation loop, the program continuously runs the trade() method of the trader and tracks the number of stocks available in the selling queue.

In my simulation, I've extended the Trader class to include more dynamic features. These features involve adding new agents, managing the total stock of the company, and simulating ongoing market activity through trades and new buying offers. These additions aim to create a more realistic simulation of a stock market environment.

...

After some inspection, I realized that this stock price was not actually random but slowly increasing.

I run the code for millions of iterations and tried again and again and saw that it was always increasing. For this code to be random, it sometimes should have end up lower than the initial price. Therefore, I tried to change the code, so that I could get rid of that upward momentum.

First version of the code, I was calling the trade function when any offer was put in the system. It was creating new agents every time a transaction couldn't be made. But this was just creating too many agents and most of the transactions were happening between same investor which was not ideal and if run some time, it was giving nonsense results.

3.3 Explanation of the Third Version

3.3.1 Agent Class (Agent)

In the Agent class, I've added two new counters, bought and sold, to keep track of the stocks an agent has bought and sold. When an agent buys stocks from another agent using the buy() method, the amount of stocks bought is added to the bought counter. Similarly, when an agent sells stocks to another agent using the sell() method, the amount of stocks sold is added to the sold counter. This helps us track the trading activity of each agent.

3.3.2 Trader Class (Trader)

I've made several changes and added new functions to the Trader class for a more sophisticated simulation

- Trade Loop: Inside the trade() method, I've introduced a loop that continues trading as long as the buying offer price is higher than or equal to the lowest selling offer price. This ensures that trades are matched properly and accounts for varying prices.
- Last Trade Price: Within the trade loop, I've included an update to the lastprice property. This property stores the price of the last successful trade. This information helps us keep track of the latest trade price.
- Updating Offers: I've introduced an update() function that takes a list of agents participating in the simulation. This function goes through each agent and checks their bought and sold counters. If an agent has bought stocks, a new selling offer is added to the selling queue using a random price within a certain range. Similarly, if an agent has sold stocks, a new buying offer is added to the buying queue.
- Price Regulation: I've added a regulate() function. This function adjusts the prices of the offers to maintain a balanced market. If the highest buying price is lower than the lowest selling price, the function randomly chooses to either increase the prices of the buying offers or decrease the prices of the selling offers.

3.3.3 StockMarketSimulation Class (StockMarketSimulation)

In the main simulation loop, I've included the following steps

- Call the regulate() function to adjust prices based on market conditions.
- Call the trade() function to perform trades and match buying and selling offers.
- Call the update() function to adjust offers for agents based on their trading history.

The loop runs continuously with a 1-second pause between iterations, creating a dynamic and evolving stock market simulation.

These changes and new functions add more realism to the simulation by simulating price adjustments, tracking trade activity for each agent, and dynamically updating buying and selling offers. This enhanced simulation captures the complex nature of a real stock market environment.

This way my code first regulates the prices so that the trades can be made, does all the trades so that there is again a gap between two offer lists, and updates the lists which is basically selling the items that have been bought and buying with balances bigger then zero so that we don't run out of offers.

Now, stock price fluctuates around the initial price and when I run the program many times and stop at some iteration of the while loop in main, I get random prices, not only bigger prices then initial price which was the problem with all the earlier versions, I get prices that are bigger and smaller which was what I wanted. Since we always have offers in queues, we don't need to create new agents, only one agent is enough, it can buy and sell to itself.

Now, I will focus on putting this function in a web server or something similar. I want to add a database, add some more companies to trade, add a graph of the company stock price and a table in website, and some functionalities like changing the starting stock and balance of the company, adding new agents with balances that can be selected, adding buying offer as an user and see how our balance is doing, and so on. For that I need to learn how to use Spring Boot.

...

Spring Boot is an open-source Java-based framework that simplifies the development of standalone, production-grade applications with minimal configuration[2]. It's part of the larger Spring ecosystem, which provides tools and libraries for building enterprise-level Java applications.

Spring Boot is designed to make it easier to set up and develop Spring applications by providing default configurations and sensible conventions. It includes a range of features and capabilities that streamline the development process, including:

1. Auto-Configuration: Spring Boot automatically configures various components based on the project's dependencies, reducing the need for manual configuration.

2. Standalone: Spring Boot applications can be run as standalone JAR files, which include an embedded web server, eliminating the need for deploying to a separate application server.

3. Opinionated Defaults: Spring Boot comes with sensible default configurations and opinions about best practices, which helps developers get started quickly while maintaining flexibility for customization.

4. Spring Boot Starters: These are pre-configured templates that provide the necessary dependencies and configurations for various application types, such as web applications, data access, messaging, etc.

5. Production Readiness: Spring Boot offers tools and features for monitoring, metrics, and health checks, making it easier to build production-ready applications.

6. Microservices Support: Spring Boot works well for building microservices-based architectures by providing features like Spring Cloud for distributed system patterns.

7. Easy Testing: Spring Boot supports testing with various tools and frameworks, making it simpler to write unit and integration tests.

8. Embedded Servers: Spring Boot supports various embedded web servers like Tomcat, Jetty, and Undertow, allowing you to choose the one that fits your application's needs.

9. Externalized Configuration: Spring Boot allows you to externalize configuration properties, making it easier to change application behaviour without modifying the code.

Spring Boot is popular in the Java ecosystem due to its ability to accelerate development and reduce boilerplate code. It's widely used for building a wide range of applications, from simple RESTful APIs to complex enterprise systems.

Now my program is running on springboot and using mysql database to store the data. It displays a chart for seeing the price change and has a add agent button for adding an agent with buying offer and see the changes immediately through the chart and current price header.

Here is an example run of my program:

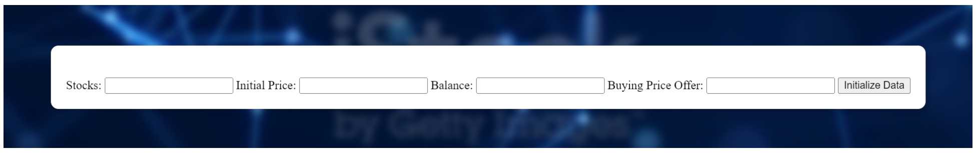
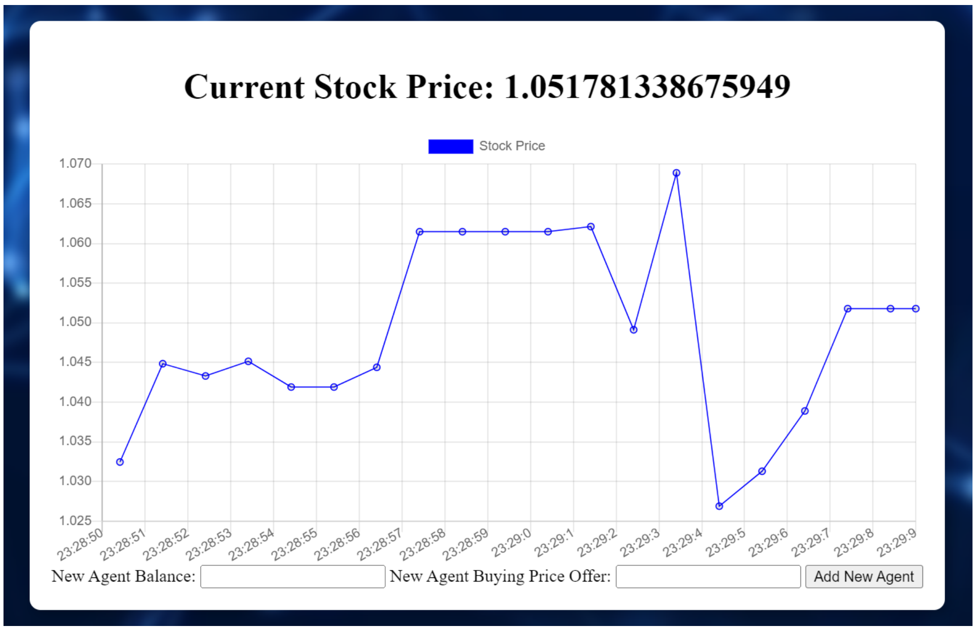


Figure 3: Pop up for input entry

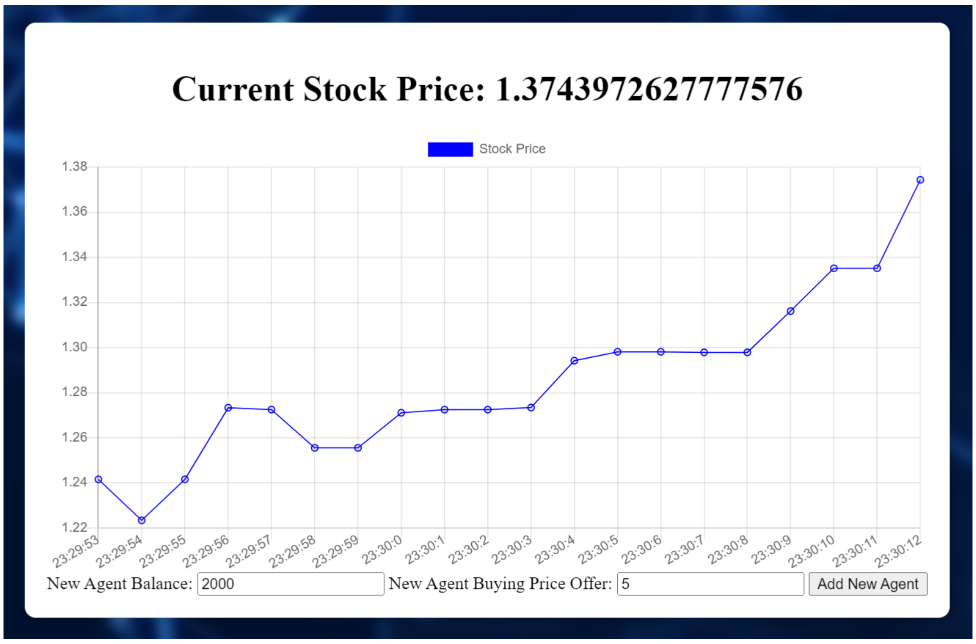
As it can be seen from Figure 3, user can enter the starting values for the simulation. Here we are initializing the first company for the stock market. Initial price was set as 1 dollar. buying offer was 1.05 dollars to start the market.



New Agent Balance:
New Agent Buying Price Offer:

Figure 4: Stock chart of added company

As it can be seen from Figure 4, after entering the values, the simulation is running and it is randomly fluctuating around the starting price which was set as 1.05. It goes down and up, but it will stay at the near of the starting price, as long as we do not add another user to manipulate the current price. New agent can be added by pressing the button at Figure 4.



New Agent Balance: 2000
New Agent Buying Price Offer: 5

Figure 5: Stock chart after new agent

As shown in Figure 5, after the user enters an offer that is higher (5) than the current price (around 1.05), fluctuation is gone and the stock price increases over time as it would happen in real life. More agents can be added to decrease or increase the stock price. One can experiment with different input values for number of stocks and starting price and observations can be made for further analysis.

3.4 About Random Number Generating Function

My class RandomNumberGenerator uses Box-muller transformation to output normally disturbed random numbers. The Box-Muller transformation is a mathematical method used to generate random numbers with a Gaussian or normal distribution[3]. It's particularly useful when you need random values that are distributed around a mean (average) value, such as when simulating random stock prices or any situation where natural variation follows a bell-shaped curve.

Here is the detailed explanation of the transformation:

1. **Start with Uniform Random Numbers:** The Box-Muller transformation begins with two independent random numbers that are uniformly distributed between 0 and 1. These uniform random numbers are typically generated using a random number generator.

2. **Transform to Polar Coordinates:** The next step involves converting the uniform random numbers into pairs of random numbers in polar coordinates. The transformation from Cartesian (rectangular) coordinates to polar coordinates is as follows:

- Let u_1 and u_2 be the two independent uniform random numbers.
- Calculate r as the square root of $-2\ln(u_1)$, where ln is the natural logarithm. This value r follows a chi-squared distribution with two degrees of freedom.
- Calculate θ as $2\pi u_2$, where π is the mathematical constant pi (approximately 3.14159265).

3. **Convert to Cartesian Coordinates:** After obtaining r and θ in polar coordinates, you can transform them into pairs of random numbers in Cartesian coordinates, which are normally distributed:

- Calculate z_0 as $r\cos(\theta)$. This is a normally distributed random number with a mean of 0 and a standard deviation of 1.
- Calculate z_1 as $r\sin(\theta)$. z_0 and z_1 are two independent normally distributed random numbers.

4. **Scaling and Shifting:** If you want normally distributed random numbers with a specific mean (μ) and standard deviation (σ), you can scale and shift the z_0 and z_1 values as follows:

- Normal random number x with mean μ and standard deviation σ can be calculated as: $x = \mu + (z_0\sigma)$.

The result is that you have generated pairs of random numbers that follow a Gaussian or normal distribution with a mean of μ and a standard deviation of σ. This is useful for simulating random data that conforms to a bell-shaped curve, which is common in various real-world scenarios, including financial modeling and statistics.

4 CONCLUSIONS

During my internship, I learned to develop web applications using Java and the Spring Boot framework. I gained experience in database management, software architecture and design, algorithm development, and continuous integration through GitHub.

Furthermore, I gained hands-on experience in user interface development. I had the opportunity to design and create a web-based user interface for my stock market simulation, honing my skills in web development and user experience design.

My project was a practical foray into simulating a real-world stock market environment. This endeavor provided insights into the intricacies of financial markets, including stock trading dynamics, market behavior, and the factors that influence stock prices. It was a valuable lesson in understanding how real financial markets function and respond to various factors.

This project can be further expanded by adding more stocks, adding options like selling at a desired price and so on, and could be used as an actual stock market simulation for practicing before entering a real stock market.

5 REFERENCES

[1] Türk Ekonomi Bankası (TEB), ”Tarihçe”, [Online]. Available: `https://www.teb.com.tr/teb-hakkinda/tarihce/`, [Accessed Sept. 6, 2023].

[2] Spring Framework, ”Spring Boot,” [Online]. Available: `https://spring.io/projects/spring-boot`, [Accessed Sept. 6, 2023].

[3] Wikipedia, ”Box–Muller Transform”, Dec. 6, 2023. [Online]. Available: `https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform`, [Accessed Sept. 6, 2023].