Comparative Analysis of Trading Strategies Using an Object-Oriented C++ Framework

Jagraj Kahlon Kirtish Meeheelaul Akshay Zine Sam Zhou

March 12, 2025

Group Name: Medallion Fund

1 Introduction

The project aims to simulate a simplified financial market environment to evaluate and compare various trading strategies. In an increasingly complex financial market, the ability to model and test trading strategies is essential. The objective of this project is to create a market simulation framework using C++, featuring an agent class capable of executing distinct trading strategies based on all available market information.

The simulation will include a limited set of assets and enable the agent to evaluate and use one of the three main trading strategies.

1. **All-in Bond Strategy**: Allocating all funds to bonds, representing a low-risk, fixed-income approach.
2. **Equal Allocation Strategy**: Distributing funds equally across all available assets, offering a naive yet diversified portfolio.
3. **Markowitz Efficient Portfolio Strategy 1**: Optimising the portfolio based on historical returns and covariances to achieve the maximum expected return for a given level of risk.
4. **Markowitz Efficient Portfolio Strategy 2**: Same idea as Markowitz Efficient Portfolio Strategy 1, with a different target return

The agent class will not only execute these strategies, but also generate performance metrics such as final wealth, returns, and variance of returns. The simulation will use historical bond returns and stock prices of a diverse list of companies: Apple, Johnson & Johnson, Procter & Gamble, Tesla, NVIDIA Corporation, Moderna, The Boeing Company, Shopify, Newmont Corporation, Simon Property Group, and The Walt Disney Company. In addition, the project is designed with scalability in mind, enabling future enhancements to include more sophisticated strategies, a greater variety of assets, and multiple interacting agents. To validate the simulation, a simplified dataset will be generated, and the calculations will be manually verified. The ultimate goal is to assess whether portfolio optimisation methods like Markowitz's provide a tangible advantage over simpler strategies under controlled conditions, setting the stage for future extensions into more realistic and dynamic market scenarios.

2 Methodology

2.1 Team Organisation and Workload Distribution

As a team, we considered several ideas, including the implementation of the limit order book and the development of a linear algebra library, but ultimately chose to compare investment strategies due to our shared interest in the topic, the clarity of its objectives, and the opportunity to also include the development of a linear algebra library as part of this project. The workload was distributed according to individual strengths.

- **Jagraj**: Designed the overall class structure and wrote the skeleton code. Implemented the agent base class and the MarkowitzSavvy agent, integrating key components, and ensuring smooth interaction between modules.
- **Akshay**: Developed the Matrix class and wrote the main simulation loop, including CSV data handling and result output.
- **Kirtish**: Implemented and evaluated tests for matrix operations and agent functions, focusing on conditions such as bankruptcy detection and portfolio normalisation.
- **Sam**: Compiled the report, generated figures including the Class Structure Diagram, and performed data analysis, providing insights into the performance of each strategy.

2.2 Overview of project

The programme runs by initialising agents who have their own investment strategy, before giving them market data for them to store and evolve their strategies around. The data stored is:

- Past bond returns as a vector
- Past asset prices as a matrix
- Previous positions of the agent as a matrix
- Wealth data of the agent as a vector
- Cumulative return of the agent as a vector

The most important functions in the project are the setup and next_step methods of the general_agent class. The setup function takes in initial asset prices and bond returns and uses them to create a time 0 portfolio and next_step takes in the most recent price and return data and uses this (alongside the previous timestep's portfolio) to update the agent. An example workflow would be:

1. Initialise an agent
2. Use setup with data for $t = 0$ to form the time 0 portfolio (all agents start with initial wealth of 1 for simplicity)
3. Use next_step with data at $t = 1$ to compute the wealth and returns and then create the portfolio for $t = 1$

4. Continually use next_step for the desired amount of time periods

For testing and analysis, we also implemented several other functions for returning key data about the agent and working out important metrics such as the variance and Sharpe ratios of the returns of the agent.

One small detail is that we used the time t bond return to work out the bond payoff for time $t + 1$ as in the real world someone purchasing a bond will typically know the interest rate for the next period.

2.3 Modelling Approach and Class Structure

The project is structured using object-oriented principles, including the following key classes:

Agent Base Class (Agent)

- **Purpose**: Acts as a base class for all investment strategies.
- **Key Methods**:
- current_wealth(): Returns current wealth of the agent.
- setup(): Initializes the agent with bond returns and asset prices.
- next_step(): Advances the simulation by one period.
- `total_return()`, `variance_return()`, `sharpe_ratio()`: Performance evaluation metrics.

Specialised Agent Classes

- **Risk Hater**: Allocates all funds to the risk-free asset (bonds).
- **Naive Investor**: Distributes funds equally across all assets.
- **Markowitz Savvy**: Implements the Markowitz Efficient Portfolio strategy, optimising portfolio weights using historical return data and matrix operations.

Utility Classes

- **Matrix Class (Matrix)**: Provides advanced matrix operations such as multiplication, inversion, and matrix slicing.
- **FileIO Class (FileIO)**: Manages reading and writing CSV files for market data and simulation results.

Figure 1 illustrates the overall project architecture and interactions among the main classes.

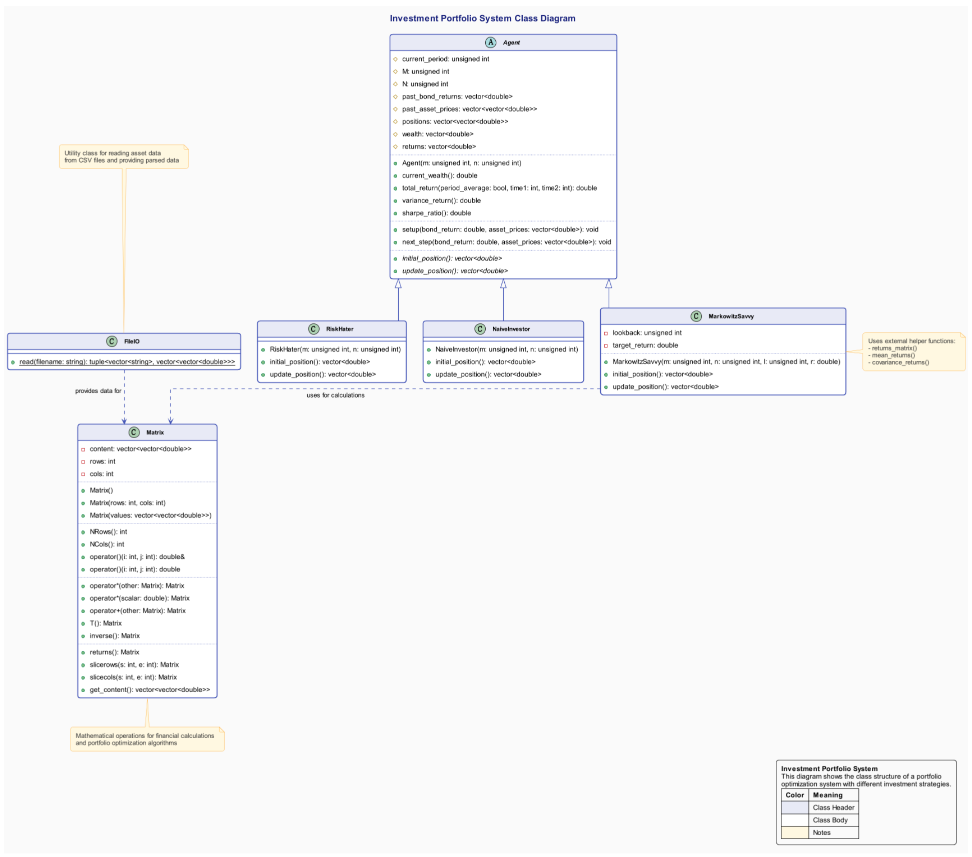
Investment Portfolio System Class Diagram
Agent
current_period: unsigned int
M: unsigned int
N: unsigned int
past_bond_returns: vector<double>
past_asset_prices: vector<vector<double>>
positions: vector<vector<double>>
wealth: vector<double>
returns: vector<double>
Agent(m: unsigned int, n: unsigned int)
current_wealth(): double
total_return(period_average: bool, time1: int, time2: int): double
variance_return(): double
sharpe_ratio(): double
setup(bond_return: double, asset_prices: vector<double>): void
next_step(bond_return: double, asset_prices: vector<double>): void
initial_position(): vector<double>
update_position(): vector<double>
Utility class for reading asset data from CSV files and providing parsed data
FileIO
read(filename: string): tuple<vector<string>, vector<vector<double>>>
RiskHater
RiskHater(m: unsigned int, n: unsigned int)
initial_position(): vector<double>
update_position(): vector<double>
NaiveInvestor
NaiveInvestor(m: unsigned int, n: unsigned int)
initial_position(): vector<double>
update_position(): vector<double>
MarkowitzSavvy
lookback: unsigned int
target_return: double
MarkowitzSavvy(m: unsigned int, n: unsigned int, l: unsigned int, r: double)
initial_position(): vector<double>
update_position(): vector<double>
Uses external helper functions:
- returns_matrix()
- mean_returns()
- covariance_returns()
provides data for
uses for calculations
Matrix
content: vector<vector<double>>
rows: int
cols: int
Matrix()
Matrix(rows: int, cols: int)
Matrix(values: vector<vector<double>>)
NRows(): int
NCols(): int
operator()(i: int, j: int): double&
operator()(i: int, j: int): double
operator*(other: Matrix): Matrix
operator*(scalar: double): Matrix
operator+(other: Matrix): Matrix
T(): Matrix
inverse(): Matrix
returns(): Matrix
slicerows(s: int, e: int): Matrix
slicecols(s: int, e: int): Matrix
get_content(): vector<vector<double>>
Mathematical operations for financial calculations and portfolio optimization algorithms
Investment Portfolio System
This diagram shows the class structure of a portfolio optimization system with different investment strategies.
Color
Meaning
Class Header
Class Body
Notes

Figure 1: Class Structure Diagram illustrating interactions between the Agent, Matrix, FileIO, and specialised agent classes.

2.4 Implementation details for MarkowitzSavvy

While the RiskHater and NaiveInvestor classes are somewhat trivial in that their position never changes, the MarkowitzSavvy class is more complex. It has the extra parameters of lookback (l) and target_return (r).

We opted to use a rolling window setup where for the first 'lookback' periods, the agent invests solely in the bond. After this, the agent invests according to Markowitz Portfolio Theory, using the previous 'lookback' asset prices to estimate the mean and covariance matrix of the returns vector.

Remember that the classical Markowitz framework generally allows short selling, which corresponds to negative portfolio weights. In practice, many investors restrict or disallow short selling, thereby confining weights to be nonnegative and changing the feasible set of portfolios. Our implementation can be adapted to either permit or prohibit short positions depending on user preference.

According to Markowitz Portfolio Theory, theoretically, any mean return is possible, so we have included the target return which you can set to any positive number and the agent will then work out a mean-variance efficient portfolio with that expected return.

Since the theory behind this relies strongly on linear algebra operations, we implemented a Matrix class capable of doing these operations, which proved incredibly effective when combined with the MarkowitzSavvy class. The formula for an efficient portfolio is given by $\bar{\pi} = (1 - \pi^T \mathbf{1}, \pi)$ where

$$\pi = \frac{\mu_{\min} - r}{Ar^2 - 2Br + C} \Sigma^{-1}(\mu - r\mathbf{1}),$$

$A = \mathbf{1}^T \Sigma^{-1} \mathbf{1}$, $B = \mathbf{1}^T \Sigma^{-1} \mu$, $C = \mu^T \Sigma^{-1} \mu$. Here, $\mu_{\min}$ represents the target return, r represents interest rate, μ and Σ are the mean and covariance matrix of the return vector and $\mathbf{1}$ is a vector of ones [4].

2.5 Testing Strategy

Comprehensive testing ensures component functionality. Tests include:

- **Matrix Operations**: Verification of multiplication, transposition, slicing, and inversion. Includes tests for handling singular matrices.

- **Agent Methods**: Tests for position normalisation, bankruptcy scenarios, and dynamic adjustments in the Markowitz portfolio.

Testing the Markowitz portfolio required a large degree of computation. We had two main test cases: one with a single asset where we explored edge cases such as bankruptcy and target return being too low and the other for multiple assets. For the former, we did all calculations by hand and worked out the exact form of the wealth and compared those to the output of the programme. The multiple asset test case would have been difficult to perform such computations manually so we opted to work out approximately what the answer should be using the NumPy library in Python. The code for these calculations is included in the root directory of the project.

3 Discussion

3.1 Simulation and Performance Evaluation

The simulation framework processes market data by reading asset prices and bond returns from CSV files using the Matrix class. Agents update portfolios each timestep. We chose to use a lookback of 180 days (about 6 months) for the Markowitz portfolio to balance the accuracy and recency of estimates for μ and Σ.

Performance metrics evaluated are Final Wealth, Cumulative Return, Volatility, and Sharpe Ratio.

Figure 2 shows sample wealth trajectories for each strategy, while Figure 3 shows cumulative returns for each strategy and Table 1 shows the performance of the implemented strategies in terms of Final Wealth, Cumulative Return, and Sharpe Ratio.

The Risk Hater strategy, allocating entirely to bonds, achieves very stable but low cumulative returns of 16%. Due to investing exclusively in bonds, we observe a Sharpe ratio of 0.0 as expected. In contrast, the Naive Investor strategy shows higher cumulative returns and moderate volatility. The two variations of the Markowitz Savvy strategy reveal contrasting outcomes: the Markowitz strategy 1 shows conservative performance, similar to bonds, due to a low target return choice, whereas the Markowitz strategy 2 achieves the highest returns of 509% but at the cost of higher volatility, highlighting the characteristic trade-off between risk and return in asset allocation. The Markowitz

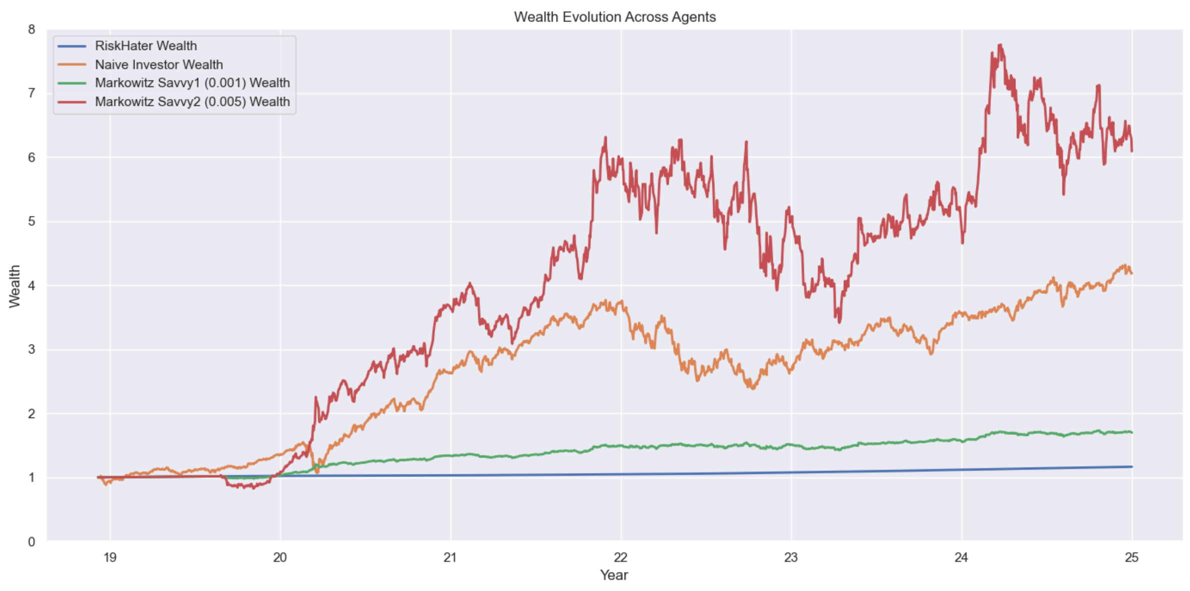

Figure 2: Wealth Evolution

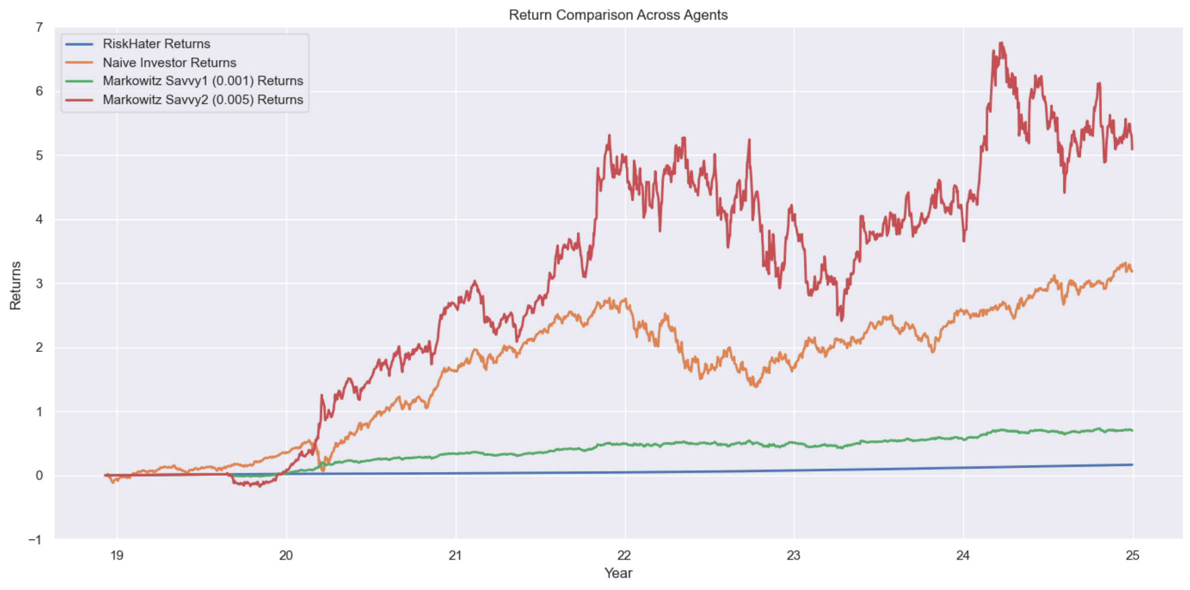

Figure 3: Cumulative Returns

Agent	**Final Wealth**	**Cumulative Return**	**Sharpe Ratio**
Risk Hater	1.16	0.16	0.00
Naive Investor	4.19	3.19	0.057
Markowitz Savvy 1	1.69	0.69	0.060
Markowitz Savvy 2	6.09	5.09	0.048
Markowitz Savvy 3	0	-1	-0.074

Table 1: Summary of simulation results for each strategy.

strategy 1 actually achieves the highest Sharpe ratio, likely due to edging out the bond returns whilst still maintaining a relatively low variance.

We also programmed a third Markowitz strategy with an even higher target return which ended up going bankrupt due to a more aggressive strategy (which likely involved heavily short-selling assets and/or the bond), but we decided to omit it from the graphs because it had a very high wealth before going bankrupt, leading to it being difficult to plot.

The comparative analysis shows the importance of strategy selection aligned with an investor's risk tolerance. The Markowitz approach, with good parameter tuning, shows great potential to outperform simpler strategies.

3.2 Strengths and Weaknesses

As for the advantages of our program's implementation, the project code is highly modular, which makes it easy to add new agents or even change the way the code behaves on a larger scale. Furthermore, the code can be used as a basis for back-testing more advanced trading strategies such as using agents with reinforcement learning. However, the high level of modularity can also lead to complexity and potential inefficiencies in scenarios such as high-frequency trading (HFT).

4 Conclusion

This project compares three trading strategies using an object-oriented C++ framework. The primary goal was to evaluate portfolio performance via simulation, demonstrating that an effective OO design and comprehensive testing are capable of simulating portfolio dynamics in mathematical finance. The main findings are:
The Risk Hater strategy: minimises risk but results in lower cumulative returns.
The Naive Investor strategy: based on equal weighting, offers balanced returns with moderate volatility.
The Markowitz Savvy strategy: with careful parameter tuning, this asset allocation approach offers significant potential for superior risk-adjusted returns.
Future work will extend the simulation framework by improving the Markowitz technique and adding additional market factors, such as those identified in the Fama-French model[3].

Use of Generative AI

Our group used ChatGPT and code from previous assignments to refine the process of reading and writing files, due to unfamiliarity with this process. Some members of our group also used Microsoft Copilot to speed up typing larger quantities of code.

References

[1] Lecture Notes on C++ Programming and Object-Oriented Design.

[2] Markowitz, H. (1952). Portfolio Selection. *Journal of Finance*, 7(1), 77–91.

[3] Fama, E. F., & French, K. R. (1993). Common risk factors in the returns on stocks and bonds. *Journal of Financial Economics*, 33(1), 3–56.

[4] David Hobson (2023). Lecture notes for Introduction to Mathematical Finance, Ch2.