Genetic algorithm for Portfolio Selection

C++

UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
MASTER 2 MMMEF

Group:

AIT ELMENCEUR Ilyes

YOKAMAMOHARAN Birawin

SAINT ROMAIN Axel

Professors: VANIER Sonia / SPENCER TRINDADE Renan

MMMEF

Université Paris 1 Panthéon Sorbonne

Contents

1 Introduction

This project endeavors to harness the power of genetic algorithms for the purpose of financial portfolio construction, specifically honing in on the optimization of the Sharpe ratio. The diverse set of financial assets under consideration includes Marathon Digital Holdings (MARA), Tesla (TSLA), NIO Inc. (NIO), Advanced Micro Devices (AMD), oFi Tecnologie (SOFI), Riot Blockchain (RIOT), Intel Corporation (INTC), Apple Inc. (AAPL), Ford Motor Company (F), Pfizer Inc. (PFE), Palantir Technologies (PLTR), and AT&T Inc. (T). The focal point of the optimization is to achieve portfolios that not only maximize returns but also minimize volatility, aligning with the principles of modern portfolio theory. The Sharpe ratio, a pivotal metric quantifying risk-adjusted returns, serves as the guiding light for evaluating and refining portfolio performance. This report provides a detailed account of the implementation of this portfolio optimization using a genetic algorithm, predominantly leveraging C++ with supplementary support from Python.

2 Research Question

2.1 Problem/Objective

This project addresses the fundamental challenge of constructing portfolios comprising both risky and risk-free financial assets. With a predetermined risk-free asset boasting a known return and zero volatility, the overarching objective is to craft portfolios that strike an optimal balance, maximizing returns while keeping volatility at a minimum.

2.2 Considered Assets

The universe of considered assets spans a spectrum of twelve financial instruments, encompassing Marathon Digital Holdings (MARA), Tesla (TSLA), NIO Inc. (NIO), Advanced Micro Devices (AMD), oFi Tecnologie (SOFI), Riot Blockchain (RIOT), Intel Corporation (INTC), Apple Inc. (AAPL), Ford Motor Company (F), Pfizer Inc. (PFE), Palantir Technologies (PLTR), and AT&T Inc. (T). Each of these assets contributes to the diverse landscape explored during the portfolio optimization process.

2.3 Performance Indicator

At the heart of our methodology is the Sharpe ratio, a pivotal metric derived from modern portfolio theory that encapsulates the essence of constructing efficient portfolios. This criterion serves as the objective function within the genetic algorithm and guiding the optimization process across different generations. Representing the slope of the Capital Market Line (CML), the Sharpe ratio plays a central role in assessing the risk-adjusted returns of portfolios.

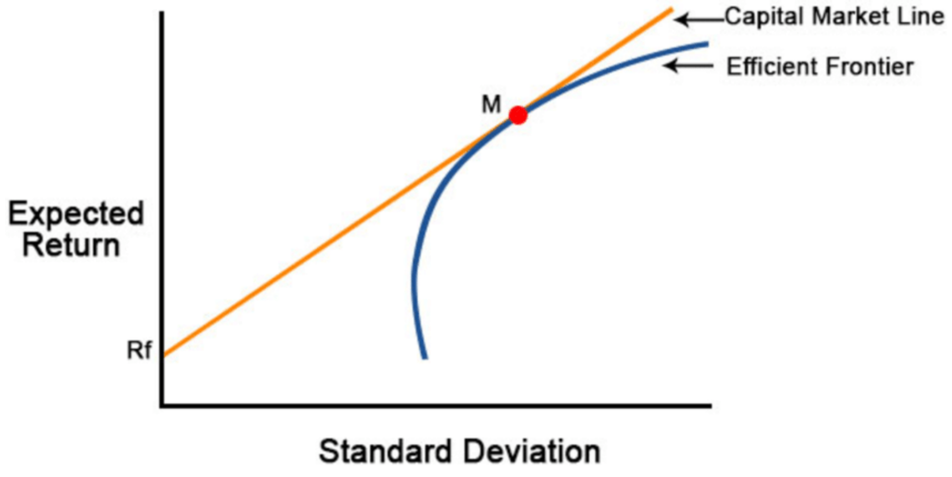


Figure 1: Illustration of the Capital Market Line & Efficient Frontier

The CML, an integral concept in modern portfolio theory, serves as the efficient frontier connecting the risk-free asset (typically placed on the expected return axis due to its zero variance) with the tangency point (M) on the upper branch of the hyperbola that represents all feasible portfolio combinations. Portfolio M, comprising risky assets, combined with the risk-free asset, enables the creation of efficient portfolios that dominate others. The maximization of the Sharpe ratio, and consequently the slope of the CML, becomes a crucial objective in constructing portfolios with optimal risk-adjusted returns. The equation of the CML is given by

$$E[R_p] = R_f + \frac{E[R_M] - R_f}{\sigma(R_M)} \cdot \sigma(R_p)$$

Where $E[R_p]$ stands for the expected return of the portfolio comprising the risk free asset and the risky assets, R_f is the risk free asset, $E[R_M]$ is the expected return of the portfolio comprising the risky assets only, $\sigma(R_M)$ is the volatility of the risky portfolio and $\sigma(R_p)$ is the volatility of the overall portfolio.

While various indicators could be considered for portfolio optimization, this project focuses solely on maximizing the Sharpe ratio, simplifying the optimization process. By honing in on this key metric, we seek to create portfolios that strike an optimal balance between risk and return, aligning with the principles of modern portfolio theory.

2.4 Genetic Algorithm

To navigate the intricacies of portfolio optimization, we employed a genetic algorithm. With this algorithmic methodology, each portfolio is represented as a chromosome, with each financial asset corresponding to a gene. It employs selection, crossover, and mutation operations to systematically evolve portfolios across successive generations. The Sharpe ratio, calculated for each portfolio at every generation, guides the selection of the most promising portfolios for subsequent reproductive steps. Chosen for its capacity to navigate diverse solution spaces, the genetic algorithm provides a structured and efficient approach to optimizing portfolios over time.

3 Implementation

3.1 C++ Code structure

The project's codebase is meticulously organized, adhering to a structured approach where each class is accompanied by both a ".hpp" (header) file and a corresponding ".cpp" file. This organization promotes enhanced code readability and maintainability.

At the core of the project, "**main.cpp**" serves as the central entry point, orchestrating the optimization process through the "**GeneticAlgorithm**" class. This class seamlessly integrates genetic algorithm principles, employing operations such as selection, crossover, and mutation to systematically evolve portfolios across successive generations.

Within this framework, the "**Portfolio**" class represents the financial portfolios generated by the algorithm, encapsulating the assigned weights for each financial asset. Simultaneously, the "**Stock**" class serves as a structured container for financial asset data, providing a standardized representation for individual assets.

The "**StockLoader**" class plays a pivotal role in managing the loading of financial data, reading standardized CSV files obtained through the Python script. This class efficiently utilizes methods and instances from the "Stock" class to organize and structure financial data for subsequent analysis.

Moreover, the "**DataUtils**" class offers essential statistical utilities and serves as a base class for both "Stock" and "Portfolio." The hierarchical relationship is further exemplified as both "Stock.cpp" and "Portfolio" inherit from "DataUtils," enabling the utilization of fundamental statistical functionalities.

3.2 UML Diagram

For a holistic understanding of the code architecture and the intricate interactions between classes, a Unified Modeling Language (UML) diagram provides a visual representation, illustrating relationships and dependencies within the project.

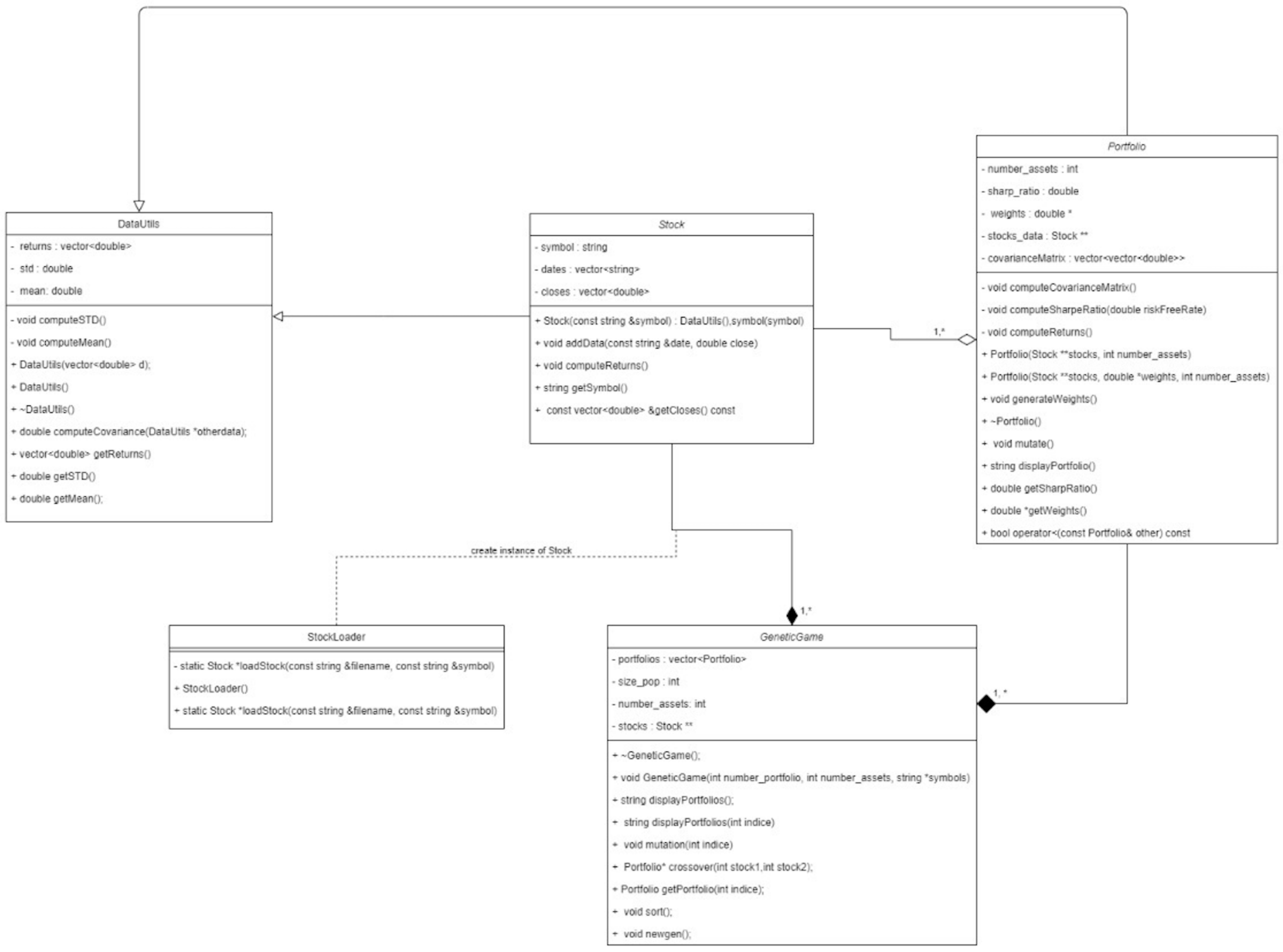

DataUtils
- returns : vector<double>
- std : double
- mean: double
- void computeSTD()
- void computeMean()
+ DataUtils(vector<double> d);
+ DataUtils()
+ ~DataUtils()
+ double computeCovariance(DataUtils *otherdata);
+ vector<double> getReturns()
+ double getSTD()
+ double getMean();
Stock
- symbol : string
- dates : vector<string>
- closes : vector<double>
+ Stock(const string &symbol) : DataUtils(),symbol(symbol)
+ void addData(const string &date, double close)
+ void computeReturns()
+ string getSymbol()
+ const vector<double> &getCloses() const
Portfolio
- number_assets : int
- sharp_ratio : double
- weights : double *
- stocks_data : Stock **
- covarianceMatrix : vector<vector<double>>
- void computeCovarianceMatrix()
- void computeSharpeRatio(double riskFreeRate)
- void computeReturns()
+ Portfolio(Stock **stocks, int number_assets)
+ Portfolio(Stock **stocks, double *weights, int number_assets)
+ void generateWeights()
+ ~Portfolio()
+ void mutate()
+ string displayPortfolio()
+ double getSharpRatio()
+ double *getWeights()
+ bool operator<(const Portfolio& other) const
1,*
create instance of Stock
1,*
StockLoader
- static Stock *loadStock(const string &filename, const string &symbol)
+ StockLoader()
+ static Stock *loadStock(const string &filename, const string &symbol)
GeneticGame
- portfolios : vector<Portfolio>
- size_pop : int
- number_assets: int
- stocks : Stock **
+ ~GeneticGame();
+ void GeneticGame(int number_portfolio, int number_assets, string *symbols)
+ string displayPortfolios();
+ string displayPortfolios(int indice)
+ void mutation(int indice)
+ Portfolio* crossover(int stock1,int stock2);
+ Portfolio getPortfolio(int indice);
+ void sort();
+ void newgen();
1, *

Figure 2: Diagramme des classes

3.3 Data extraction

The critical role of data extraction is fulfilled by the Python script "datadownloader.py," which interfaces with Yahoo Finance to extract crucial information. The recorded information includes the distinctive dates corresponding to stock quotations, featuring the opening price, representing the closing value of the financial instrument; the lowest intraday price, denoting the minimum level the stock descended to during the trading day; the closing price, signifying the final valuation at the end of the trading session; the adjusted closing price, reflecting modifications for events like dividends or stock splits; and the trading volume, quantifying the number of shares exchanged during the course of the day. This comprehensive set of data spans the temporal domain from January 1, 2023, to December 31, 2023, providing a rich foundation for the analysis and optimization of financial portfolios. The script saves these information in a standardized CSV format (one for each stock). This ensures the availability of requisite data for further processing by the C++ components.

4 Explanation of the C++ Code

4.1 DataUtils

The DataUtils class is utilized for performing statistical calculations on a dataset, including the computation of mean, standard deviation, and covariance. This class is employed by both the Stock and Portfolio classes. Consequently, only calculations that are shared between these two objects are defined within the DataUtils class. For instance, the calculation of returns cannot be accommodated in this class, as it must be performed directly within the Stock and Portfolio classes. Indeed, this specific calculation varies depending on the nature of the considered object. Therefore, statistical indicator calculations that are specific to an object are directly defined within the corresponding class.

4.2 Stock

The Stock class represents a stock and encapsulates its information. It has a constructor to initialize the stock symbol and inherits functionality from the DataUtils class. The class provides methods to add data (date and close values) to the stock, compute returns, and retrieve the stock symbol. The computeReturns() method calculates returns based on the closing prices stored in the closes vector.

4.3 StockLoader

The StockLoader class provides functionality to load stock data from CSV files and create Stock objects and an array of Stock pointers. It contains two static functions. loadStock() function creates a Stock object by reading data from a CSV file. It opens the file, reads the header, and then reads data lines, extracting dates and closing prices. The extracted data is added to the Stock object, and returns are computed. The function returns the created Stock object. loadStocks() function creates an array of Stock pointers by calling the loadStock() function for each stock symbol in the input array. This dynamic allocation provides a significant memory gain. Indeed, since the Stock class is at the core of our program, (because accessed by other classes), using pointers avoids the creation of duplicates with each call.

4.4 Portfolio

This class is designed to represent an investment portfolio composed of multiple stocks. It inherits functionality from the DataUtils class. It incorporates a method, generateWeights(), responsible for randomly assigning weights to each asset within the portfolio. Additionally, it features methods for computing diverse financial metrics, including returns, covariance matrix, and the Sharpe ratio. A noteworthy function, mutate(), introduces random modifications to the weight of an asset within the portfolio. Furthermore, the integration of a superior order comparator facilitates the evaluation of two portfolios based on their Sharpe ratios, serving as the fitness parameter in our genetic algorithm. This implementation allow us to sort vector of portfolios regarding their Sharpe ratio.

4.5 GeneticAlgorithm

The GeneticGame class represents a vector of portfolios, enabling genetic manipulations on portfolios in each generation. Once again, this class utilizes dynamic allocation for memory optimization. With each new generation, the destructor frees dynamically allocated memory for the Portfolio objects in Portfolios.

There are two display methods: the first one displays all portfolios in the population, while the second one shows the portfolio at the specified index. The mutation() method encapsulates mutate() method defined in the Portfolio class, allowing random modifications to the weights of assets within a portfolio. The crossover() method creates a new portfolio by combining the weights of two existing portfolios. Weights are selected up to a random point, and the remaining weights are taken from the second portfolio. Subsequently, the weights are normalized. The sort() method sorts the population of portfolios in ascending order based on their Sharpe ratio. The newGen() method sorts the portfolios, retains the top-performing half (elitism), performs crossovers among the remaining portfolios, and then applies mutations to the entire population.

5 Results

As mentioned in the preceding sections of this report, we considered 12 assets for constructing our portfolios. Regarding the hyperparameters of the genetic algorithm, we arbitrarily set the number of generations and portfolios built at each generation to 100.

Concerning the evolution of the Sharpe ratio for the best portfolios across generations, we observe that, at the first generation, the best portfolio demonstrated a ratio of approximately 0.16, compared to approximately 0.18 at the last generation. The ratio thus evolved by 16% (+0.02) over the 100 generations. However, the left graph in Figure 3 (depicted by the blue curve), illustrating the evolution of this ratio for the best portfolios across generations, displays a rising and concave curve. We notice that the slope of this curve is steep between the first and the fortieth generation, becoming gentler between the fortieth and the last generation. Indeed, the 16% evolution of this ratio is primarily attributable to the initial forty generations (approximately 14% evolution during these initial forty generations). This is why we decided not to increase the number of portfolios and generations, as the performance gain obtained would have been marginal.

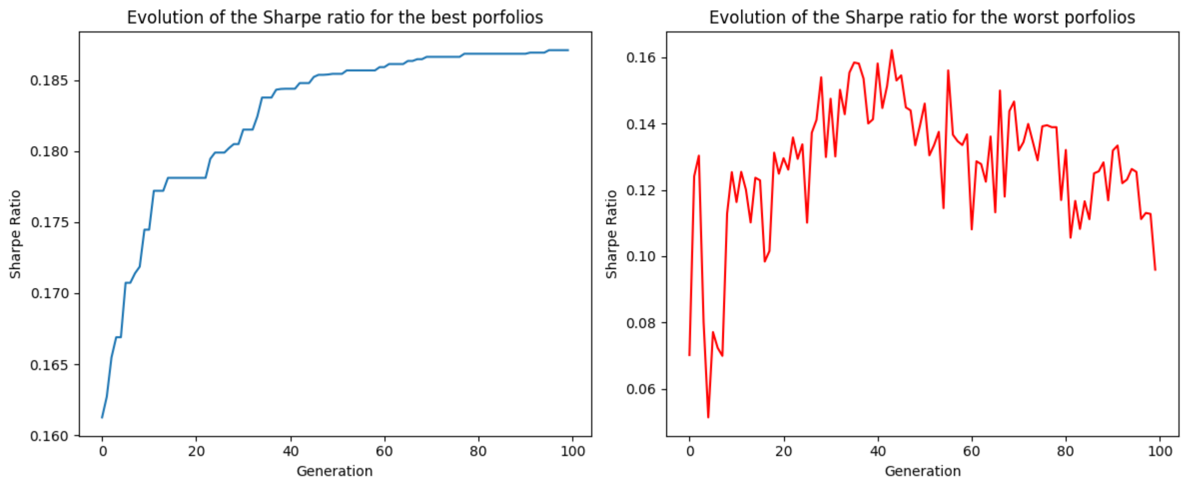
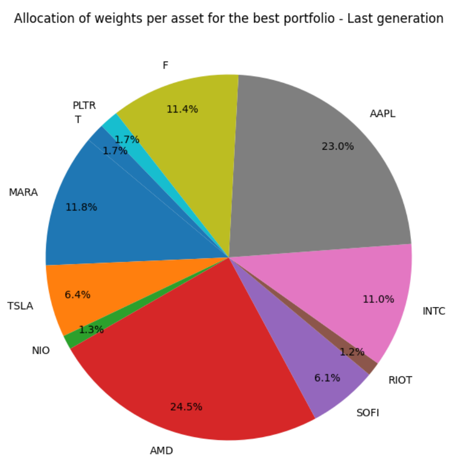


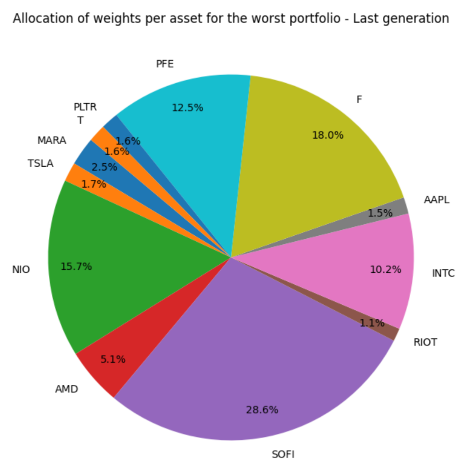
Figure 3: Evolution of Sharpe Ratios for the Best and Worst Portfolios Across Generations

Regarding the evolution of the Sharpe ratio for the worst portfolios across generations, we observe that, at the first generation, the worst portfolio demonstrated a ratio of approximately 0.07, compared to approximately 0.1 at the last generation. The ratio thus evolved by 37% (+0.03) over the 100 generations. In terms of percentages, this evolution is approximately 2.3 times higher than that of the best portfolios. This can be explained by the fact that the Sharpe ratios of the worst portfolios are, on average, more than 2 times lower than the Sharpe ratios of the best portfolios. However, the right graph in Figure 3 (depicted by the red curve), illustrating the evolution of this ratio for the worst portfolios across generations, shows substantial variations in Sharpe ratio levels across generations. Furthermore, the trend of this curve becomes decreasing from around the fortieth generation (increasing trend in the initial forty generations). The maximum Sharpe ratio for the worst portfolios is reached at the 43rd generation, approximately 0.16, closely approaching the average Sharpe ratio level of the best portfolios, and marking a 131% increase compared to the Sharpe ratio of the worst portfolio in the first generation. Therefore, it is possible to assert that the maximum performance of the genetic algorithm was achieved around the fortieth generation and then plateaued.

The pie charts depicted in Figures 4 illustrate the asset allocation in the best and worst portfolios at the last generation. In the best portfolio, Apple Inc. (23.0%) and Advanced Micro Devices (24.5%) are the most heavily weighted assets. Conversely, for the worst portfolio, Apple Inc. (1.5%) and Advanced Micro Devices (5.1%) are among the least weighted assets. The most heavily weighted assets are SoFi Technologies (28.0%) and Ford Motor Company (18.0%).



(a) Asset Allocation of Weights for the Best Portfolios - Last Generation



(b) Asset Allocation of Weights for the Worst Portfolios - Last Generation

Figure 4: Comparison of asset allocation for the best and worst portfolios - Last generation

In order to assess the efficiency of our C++ code in terms of execution time, we compared its execution time based on the number of assets considered for portfolio construction. The graph depicted in Figure 5 illustrates the program's execution time as a function of the number of assets considered. The curve of this graph is slightly below the diagonal line, indicating that the execution time is not entirely proportional to the number of assets considered. In other words, the execution time increases slightly more steeply than the number of assets considered

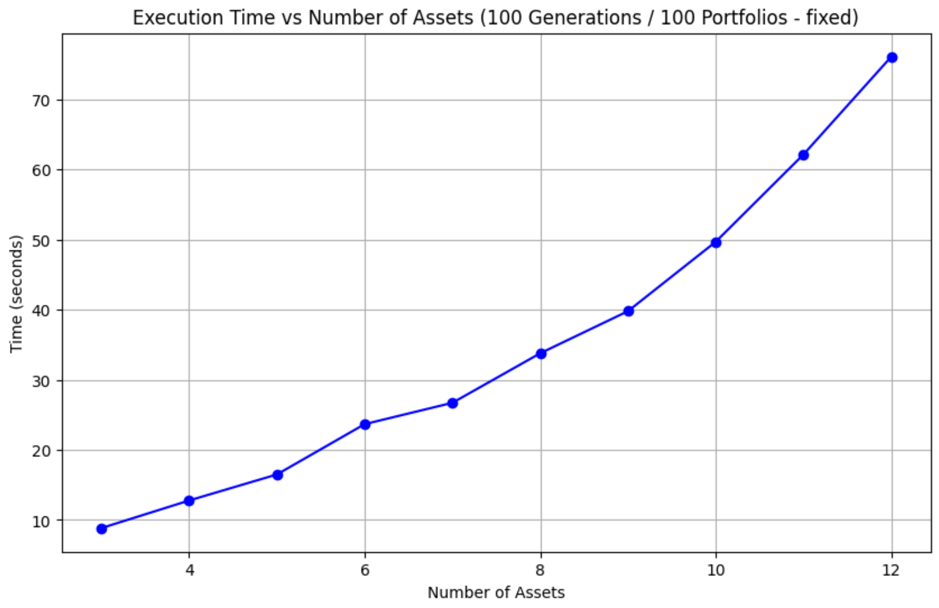


Figure 5: Execution Time Variation with the Number of Considered Assets

6 Potential Enhancements for C++ Code

Execution time in a genetic algorithm is primarily dependent on the computation of fitness. In our code, there is potential for a more efficient implementation of fitness calculation. Our fitness calculation involves the computation of mean, variance, and the covariance matrix. It is in these specific calculations that we have implementations incurring certain algorithmic costs, which could be enhanced through various methods. For instance, we could optimize by calculating mean and variance in a single loop instead of employing two separate loops as currently implemented in our project. As for the covariance matrix, we could explore register manipulation rather than using memory spaces dedicated to a single variable. In C++, this translates to leveraging Streaming SIMD Extension (SSE) libraries, offering potential optimizations. Another area for improvement would involve implementing a mechanism to halt generations when the genetic algorithm converges.

7 Conclusion

To conclude, we have implemented a genetic algorithm in this project applied to a portfolio vector. In each generation, methods such as mutation, crossover, and elitism have allowed us to obtain increasingly efficient portfolios. In our case, we observe that for 12 assets, the algorithm converges after 43 generations. It is crucial to underline that the dynamic memory allocation for stock vectors and portfolio vectors enables optimized memory management for more efficient execution on the machine.