Imperial College London

IMPERIAL COLLEGE BUSINESS SCHOOL

COMPUTATIONAL FINANCE WITH C++

Markowitz Model & Rolling Window Back-Testing

RAKESH BALI

02478023

June 3, 2024

1 Software Structure

Git Repository Link
https://github.com/rakeshbali31/ComputationalFinanceCoursework

1.1 Classes Description

1.1.1 CsvReader

Purpose: Reads CSV files and stores the data in a matrix.
Key Methods:

- `readCsv(const std::string& filename, Matrix& matrix)`: Reads a CSV file and stores the data in a matrix.

1.1.2 Algorithm

Purpose: Contains algorithms for mathematical computations, like the conjugate gradient method.
Key Methods:

- `conjugateGradient(const Matrix& Q, const Vector& b, const Vector& x0, double tolerance)`: Solves a linear system using the conjugate gradient method.

1.1.3 AssetCharacteristics

Purpose: Provides methods to calculate statistical characteristics of assets.
Key Methods:

- `calculateMeans(const Matrix& assetReturns, int startRow, int endRow)`: Calculates the mean returns of assets.
- `calculateCovariance(const Matrix& assetReturns, const Vector& means, int startRow, int endRow)`: Calculates the covariance matrix of asset returns.

1.1.4 PortfolioSolver

Purpose: Solves for the optimal portfolio weights given the asset returns and target return.
Key Methods:

- `solver() const`: Computes the optimal portfolio weights.
- `setAssetReturns(const Matrix& matrix)`: Sets the asset returns.
- `setTargetReturn(double target)`: Sets the target return.
- `setRange(int start, int end)`: Sets the range for in-sample data. This sets the range of rows of asset returns to use for the calculation.
- `getAssetReturns() const`: Gets the asset returns.
- `getTargetReturn() const`: Gets the target return.
- `getRange() const`: Gets the in-sample data range.

1.1.5 Backtest

Purpose: Performs backtesting of the portfolio using out-of-sample data.
Key Methods:

- `backtest()`: Conducts the backtest and stores the results in out-of-sample-returns and out-of-sample-variances matrices.
- `getInputData()`: Gets input data for backtesting i.e., in-sample days and out-sample days.
- `displayResults() const`: Displays the backtest results.
- `saveResultsToCSV(const std::string& filename) const`: Saves the backtest results to a CSV file.

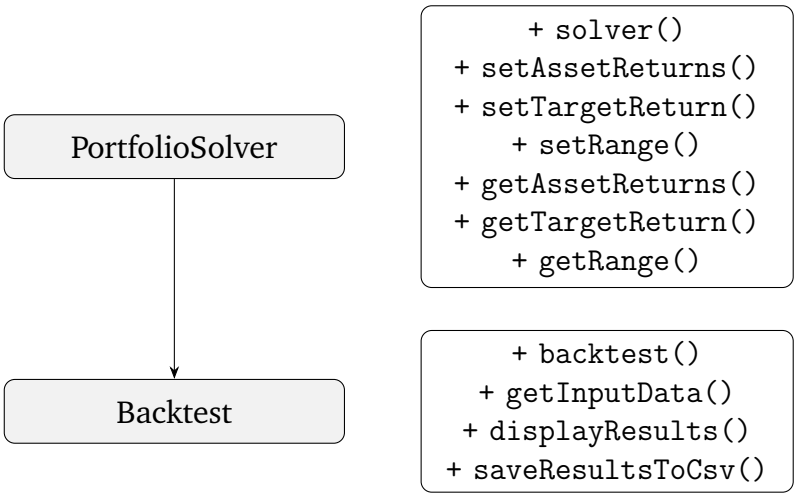
1.1.6 Matrix

Purpose: Defines a matrix and provides basic linear algebra operations.
Key Methods:

- Operator overload for matrix and vector operations.

1.2 Diagrams

1.2.1 Class Hierarchy



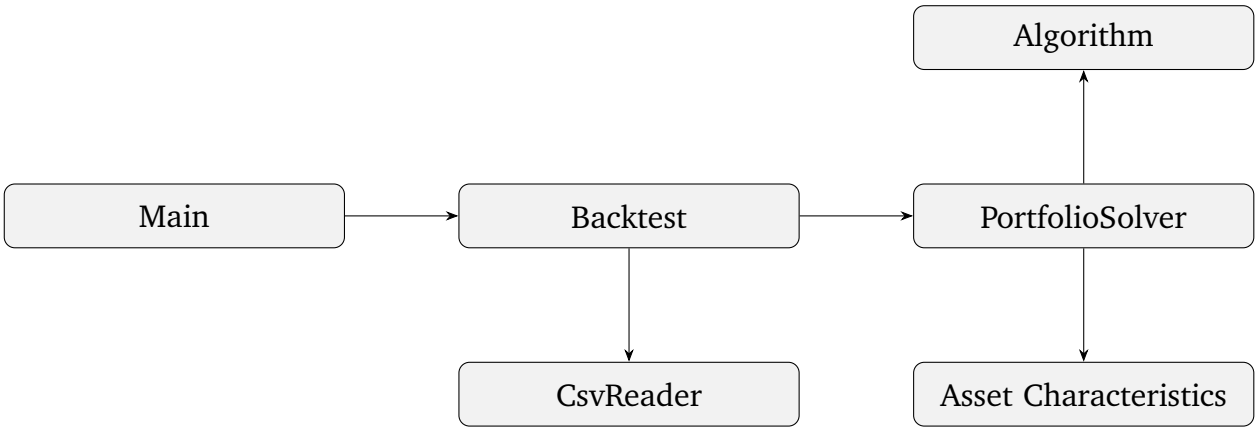
1.2.2 Relationships

- **CsvReader** is used by **Backtest** to read CSV files for input data.
- **Algorithm** provides mathematical methods used in **PortfolioSolver** for solving optimization problems.
- **AssetCharacteristics** provides methods to calculate asset statistics, which are used in **PortfolioSolver**.
- **PortfolioSolver** is the base class for **Backtest**, providing methods to solve for the optimal portfolio.

- **Backtest** extends **PortfolioSolver** to add functionality for backtesting the portfolio and also stores the backtesting results.

- The **Main** class is the entry point of the program. It initializes the **Backtest** object and coordinates the backtesting process.

1.2.3 Class Diagram



2 Evaluation

2.1 Plots

Figure 1 illustrates the return charts for 83 FTSE 1000 companies. This plot confirms the accuracy of the data for all the stocks and ensures it is free from NaN values.

Figure 2 shows the average out-of-sample returns for different target returns over multiple rolling windows. Each line represents a target return, and the x-axis denotes the rolling window index. The average OOS returns fluctuate significantly over time. Higher target returns generally correspond to higher volatility in OOS returns.

Figure 3 presents the cumulative returns for each target return over the entire backtesting period. Each line represents the cumulative performance of a target return. Cumulative returns for higher target returns (e.g., 0.1) show significant growth compared to lower target returns. There are periods of drawdowns where cumulative returns decrease, particularly noticeable in mid-windows.

Figure 4 scatter plot compares the mean returns versus variance (risk) for different target returns during the first period. Colors indicate different target returns. There is a clear positive correlation between risk (variance) and return. Higher target returns correspond to higher variance, illustrating the risk-return trade-off.

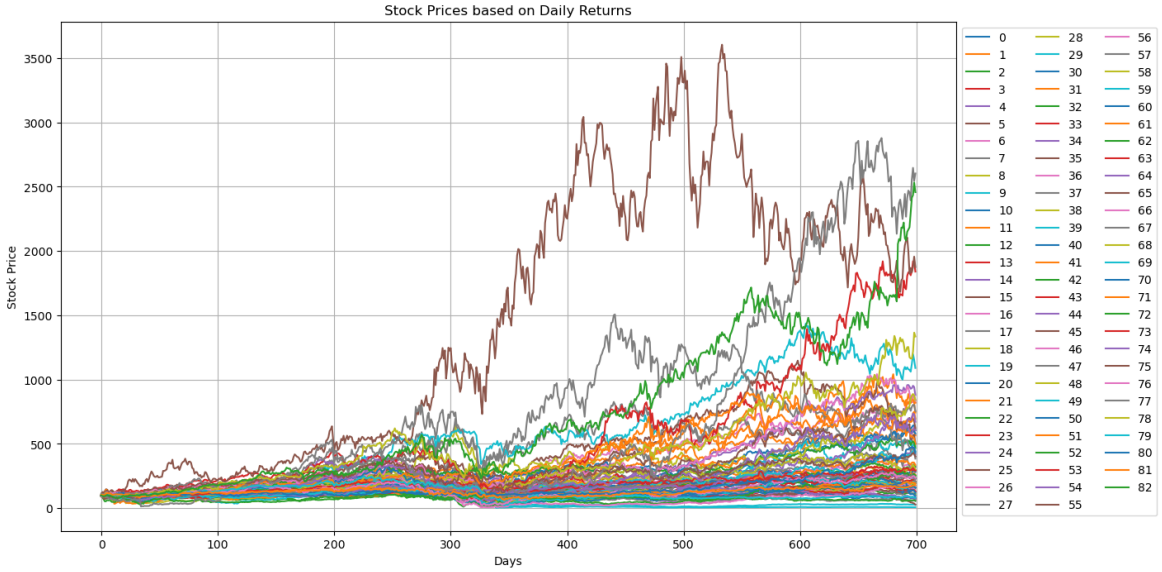


Figure 1: Stock Returns Chart

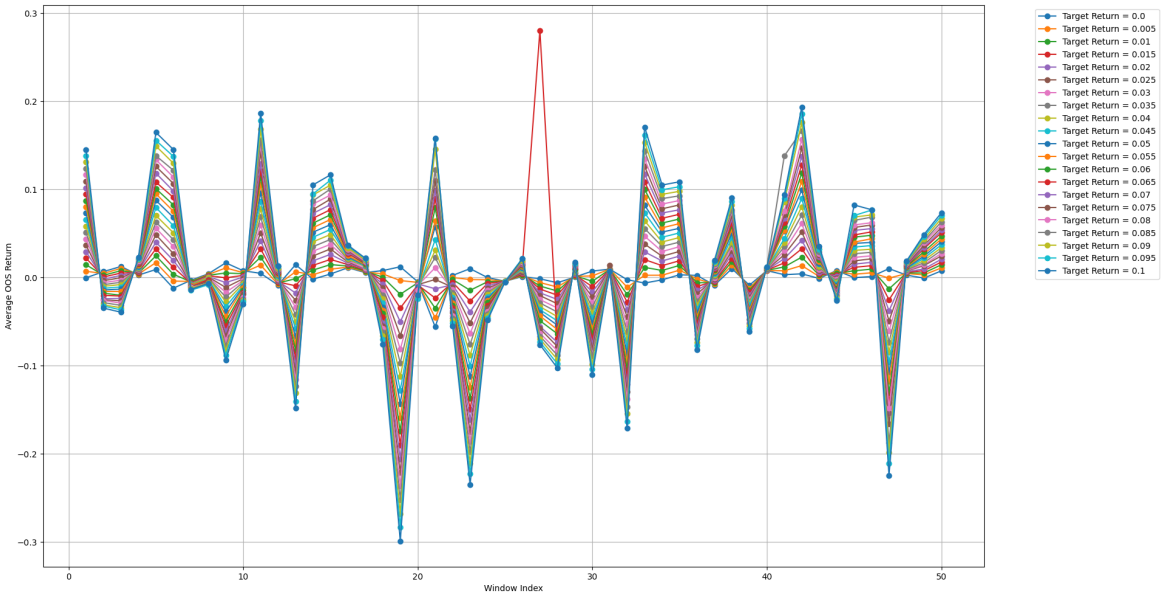


Figure 2: Realised Rolling Window Average OOS Returns for each Target

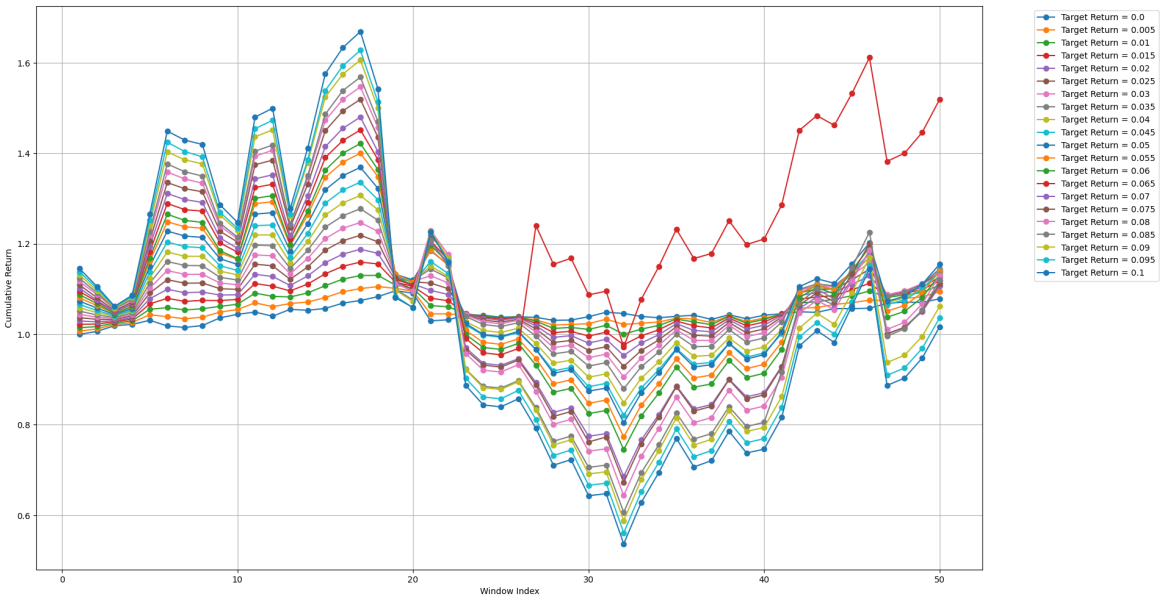


Figure 3: Cumulative Average OOS Returns for each target

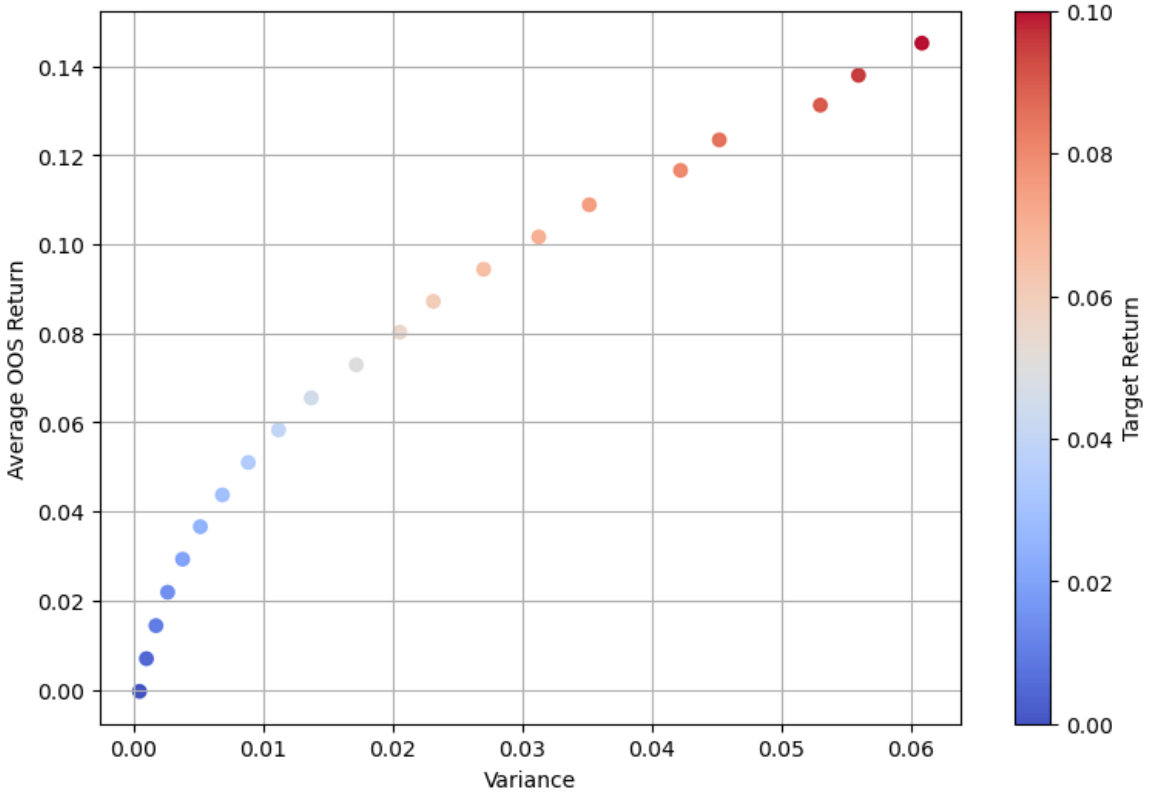


Figure 4: Risk vs Return plot for Period 1

2.2 Discussion

The evaluation of the Markowitz model, as illustrated through these figures, underscores several key points:

2.2.1 Efficient Frontier

The plots collectively illustrate the efficient frontier, highlighting the trade-off between risk and return. Portfolios on the efficient frontier provide the best possible returns for a given level of risk.

2.2.2 Optimal Portfolio

The optimal portfolio for an investor depends on their risk tolerance. Higher risk tolerance would align with portfolios targeting higher returns (and accepting higher variance), while risk-averse investors would prefer portfolios with lower target returns and variance.

2.2.3 Practical Application

The rolling window analysis shows that the performance of the Markowitz model is dynamic and sensitive to changing market conditions. Regular rebalancing of the portfolio, as demonstrated, is essential to maintain optimal performance.
In conclusion, the Markowitz model provides a robust framework for portfolio optimization, balancing the trade-off between risk and return. The figures illustrate the practical outcomes of applying this model, showcasing its strengths and areas of variability across different market conditions.

3 Code

- Main.cpp
- Backtest.cpp
- PortfolioSolver.cpp
- AssetCharacteristics.cpp
- Algorithm.cpp
- CsvReader.cpp
- Matrix.cpp

3.0.1 Main.cpp

```
#include <iostream>
#include "Backtest.h"

int main() {
    Backtest backtest;
    if(!backtest.getInputData()) {
        std::cout << "Exiting the program";
        return 0;
    }
    backtest.backtest();
    backtest.displayResults();
    backtest.saveResultsToCSV("results_python.csv");

}
```

3.0.2 Backtest.cpp

```
// Backtest.h

#ifndef BACKTEST_H
#define BACKTEST_H
#include <PortfolioSolver.h>

class Backtest: public PortfolioSolver {
private:
    int out_of_sample_period;
    Vector target_returns;
    Matrix out_of_sample_returns;
    Matrix out_of_sample_variance;

public:
    Backtest() : out_of_sample_period(10) {} // Default constructor initializing all members.
    Backtest(int out_of_sample_period, const Vector& target_returns) : out_of_sample_period(out_of_sample_period),
            target_returns(target_returns){}
    void backtest();
    int getInputData();
    void displayResults() const;
    void saveResultsToCSV(const std::string& filename) const;
};
#endif //BACKTEST_H

// Backtest.cpp
```

```
#include "Backtest.h"
#include <Algorithm.h>
#include <AssetCharacteristics.h>
#include <CsvReader.h>
#include <iomanip>
#include <iostream>
#include <fstream>


void Backtest::backtest() {
    const int num_of_days = getAssetReturns().size();
    const int in_sample_period = getEndRow() - getStartRow();

    for (const auto target_return:target_returns) {
        setRange(0, in_sample_period);
        int start = getStartRow(), end = getEndRow();
        Vector portfolioReturns;    // Local variable to store the portfolio returns for this target return
        Vector portfolioVariances;  // Local variable to store the portfolio variances for this target return
        setTargetReturn(target_return);
        while(end < num_of_days) {
            Vector weights = solver();

            Vector meanReturns = AssetCharacteristics::calculateMeans(getAssetReturns(), end, end+out_of_sample_period);
            double portfolioReturn = weights ^ meanReturns;
            portfolioReturns.push_back(portfolioReturn);

            Matrix covariance = AssetCharacteristics::calculateCovariance(getAssetReturns(), meanReturns,
                                    end, end+out_of_sample_period);
            double portfolioVariance = weights ^ (covariance * weights);
            portfolioVariances.push_back(portfolioVariance);

            end += out_of_sample_period;
            start += out_of_sample_period;
            setRange(start, end);
        }
        out_of_sample_returns.push_back(portfolioReturns);
        out_of_sample_variance.push_back(portfolioVariances);
    }

}

int Backtest::getInputData() {
    std::string filename;
    int in_sample_period;

    //std::cout << "Enter the name of the excel file: "; std::cin >> filename;
    CsvReader reader;
    Matrix matrix;
    const bool success = reader.readCsv("asset_returns.csv", matrix);
    if (!success) {
        std::cerr << "Failed to read the CSV file." << std::endl;
        return 0;
    }

    //std::cout << "Enter the target return: "; std::cin >> targetReturn;
    double start=0.0, end=0.1, step=0.005;
    for (double value = start; value <= end; value += step) {
        target_returns.push_back(value);
    }

    std::cout << "Enter the insample period: "; std::cin >> in_sample_period;
    std::cout << "Enter the outsample period: "; std::cin >> out_of_sample_period;

    // Now set the values in PortfolioSolver
    setAssetReturns(matrix);
    setRange(0, in_sample_period);
    return 1;
}
```

```
void Backtest::displayResults() const {

    for (int i = 0; i < target_returns.size(); ++i) {
        std::cout << "Printing results for target return :: " <<
        std::fixed << std::setprecision(3) << target_returns[i] <<   std::endl << std::endl;

        Vector returns = out_of_sample_returns[i];
        Vector variances = out_of_sample_variance[i];
        const int array_size = returns.size();

        // Print header
        std::cout << std::setw(10) << "Period" << std::setw(20) << "Mean Return" << std::setw(20) <<
        "Variance" << std::endl;

        for (int j = 0; j < array_size; ++j) {
            // Print data for each period
            std::cout << std::setw(10) << j + 1 << std::setw(20) << std::fixed << std::setprecision(10) <<
            returns[j] << std::setw(20) << variances[j] << std::endl;
        }
        std::cout << std::endl;
    }
}


void Backtest::saveResultsToCSV(const std::string& filename) const {
    std::ofstream outFile(filename);

    for (int i = 0; i < target_returns.size(); ++i) {
        Vector returns = out_of_sample_returns[i];
        Vector variances = out_of_sample_variance[i];
        const int array_size = returns.size();

        // Write header to file
        outFile << "Period,Mean_Return,Variance,Target_Return\n";

        for (int j = 0; j < array_size; ++j) {
            // Write data for each period to file
            outFile << (j + 1) << "," << returns[j] << "," << variances[j] << "," << target_returns[i] << "\n";
        }
        outFile << "\n";
    }
    outFile.close();
}
```

3.0.3 PortfolioSolver.cpp

```
// PortfolioSolver.h

#ifndef PORTFOLIOSOLVER_H
#define PORTFOLIOSOLVER_H

#include <Matrix.h>

class PortfolioSolver {
private:
    Matrix assetReturns; // Stores returns of various assets within a specified range.
    double targetReturn; // Target return for the portfolio.
    int startRow;        // Starting index of the row in asset returns data.
    int endRow;          // Ending index of the row in asset returns data.

public:
    // Constructors
    PortfolioSolver() : targetReturn(0.0), startRow(0), endRow(0) {} // Default constructor initializing all members.
    explicit PortfolioSolver(const Matrix& matrix, double target, int start, int end)
        : assetReturns(matrix), targetReturn(target), startRow(start), endRow(end) {}
    // Constructor to initialize all properties.
```

```
    // Setters
    void setAssetReturns(const Matrix& matrix) { assetReturns = matrix; }
    void setTargetReturn(double target) { targetReturn = target; }
    void setRange(int start, int end) { startRow = start; endRow = end; }

    // Getters
    const Matrix& getAssetReturns() const { return assetReturns; }
    double getTargetReturn() const { return targetReturn; }
    int getStartRow() const { return startRow; }
    int getEndRow() const { return endRow; }

    // Solver
    Vector solver() const;
};

#endif // PORTFOLIOSOLVER_H


// PortfolioSolver.cpp

#include "PortfolioSolver.h"
#include <Algorithm.h>
#include <AssetCharacteristics.h>


Vector PortfolioSolver::solver() const {
    const int numAssets = assetReturns[0].size();
    const auto means = AssetCharacteristics::calculateMeans(assetReturns, startRow, endRow);
    const auto covariances = AssetCharacteristics::calculateCovariance(assetReturns, means, startRow, endRow);

    // Create the augmented matrices for optimization
    Matrix Q(numAssets + 2, Vector(numAssets + 2, 0.0));
    Vector b(numAssets + 2, 0.0);
    Vector x0(numAssets + 2, 1.0); // Initial guess

    // Fill Q and b based on the model
    for (int i = 0; i < numAssets; ++i) {
        for (int j = 0; j < numAssets; ++j) {
            Q[i][j] = covariances[i][j];
        }
        Q[i][numAssets] = -means[i];
        Q[i][numAssets + 1] = -1;

        Q[numAssets][i] = -means[i];
        Q[numAssets + 1][i] = -1;
    }
    b[numAssets] = -targetReturn;
    b[numAssets + 1] = -1;

    // Solve using the conjugate gradient method
    auto weights = Algorithm::conjugateGradient(Q, b, x0, 1e-6);
    weights.pop_back(); weights.pop_back(); // Removing lambda and M parameter
    return weights;
}
```

3.0.4 AssetCharacteristics.cpp

```
// AssetCharacteristics.h

#ifndef ASSET_CHARACTERISTICS_H
#define ASSET_CHARACTERISTICS_H

#include <Matrix.h>
#include <vector>

class AssetCharacteristics {
```

```
public:
    static Vector calculateMeans(const Matrix& returns, int startRow, int endRow);
    static Matrix calculateCovariance(const Matrix& returns, const Vector& means, int startRow, int endRow);
};

#endif // ASSET_CHARACTERISTICS_H

// AssetCharacteristics.cpp

#include "AssetCharacteristics.h"

using Vector = std::vector<double>;
using Matrix = std::vector<Vector>;

Vector AssetCharacteristics::calculateMeans(const Matrix& returns, int startRow, int endRow) {
    int numAssets = returns[0].size();
    Vector means(numAssets, 0.0);

    for (int i = 0; i < numAssets; ++i) {
        for (int k = startRow; k < endRow; ++k) {
            means[i] += returns[k][i];
        }
        means[i] /= (endRow - startRow);
    }
    return means;
}

Matrix AssetCharacteristics::calculateCovariance(const Matrix& returns, const Vector& means, int startRow, int endRow) {
    int numAssets = returns[0].size();
    Matrix covariance(numAssets, Vector(numAssets, 0.0));

    for (int i = 0; i < numAssets; ++i) {
        for (int j = i; j < numAssets; ++j) {  // Start from i to fill the upper triangle
            double sum = 0.0;
            for (int k = startRow; k < endRow; ++k) {
                double diff_i = returns[k][i] - means[i];
                double diff_j = returns[k][j] - means[j];
                sum += diff_i * diff_j;
            }
            covariance[i][j] = sum / (endRow - startRow - 1);  // Using N-1 (sample covariance)
            covariance[j][i] = covariance[i][j];  // Symmetric matrix
        }
    }

    return covariance;
}
```

3.0.5 Algorithm.cpp

```
//Algorithm.h

#ifndef ALGORITHM_H
#define ALGORITHM_H

#include "Matrix.h"

class Algorithm {
public:

    static Vector conjugateGradient(const Matrix& Q, const Vector& b, const Vector& x0, double tolerance);

};

#endif // ALGORITHM_H

//Algorithm.cpp
```

```
#include "Algorithm.h"


// Function to implement the conjugate gradient algorithm
Vector Algorithm::conjugateGradient(const Matrix& Q, const Vector& b, const Vector& x0, double tolerance) {
    Vector x = x0;
    Vector r = b - (Q * x); // Using operator- for subtraction
    Vector p = r;
    double rsold = r ^ r; // Using operator^ for dot product

    for (size_t i = 0; i < b.size(); ++i) {
        Vector Qp = Q * p;
        double alpha = rsold / (p ^ Qp); // Using operator^ for dot product
        x = x + (alpha * p); // Using operator+ for addition and operator* for scalar multiplication
        r = r - (alpha * Qp); // Using operator- for subtraction and operator* for scalar multiplication
        double rsnew = r ^ r; // Using operator^ for dot product

        if (sqrt(rsnew) < tolerance)
            break;

        double beta = rsnew / rsold;
        p = r + (beta * p); // Using operator+ for addition and operator* for scalar multiplication
        rsold = rsnew;
    }
    return x;
}
```

3.0.6 CsvReader.cpp

```
// CsvReader.h

#ifndef CSVREADER_H
#define CSVREADER_H

#include <Matrix.h>
#include <string>

class CsvReader {
public:
    static bool readCsv(const std::string& filename, Matrix& matrix);
};


#endif //CSVREADER_H


// CsvReader.cpp

#include "CsvReader.h"
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>


bool CsvReader::readCsv(const std::string& filename, Matrix& matrix) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cerr << "Unable to open file: " << filename << std::endl;
        return false;
    }

    std::string line;
    while (getline(file, line)) {
        std::stringstream ss(line);
        std::string cell;
        Vector row;
```

```
        while (getline(ss, cell, ',')) {
            row.push_back(std::stod(cell));
        }
        matrix.push_back(row);
    }

    file.close();
    return true;
}
```

3.0.7 Matrix.cpp

```
// Matrix.h

#ifndef Matrix_h
#define Matrix_h

#include <vector>
using namespace std;

typedef vector<double> Vector;
typedef vector<Vector> Matrix;

Vector operator*(const Matrix& C,const Vector& V);
Vector operator*(const double& a,const Vector& V);
Vector operator+(const double& a,const Vector& V);
Vector operator+(const Vector& V,const Vector& W);
Vector operator-(const Vector& V, const double& a);
Vector operator-(const Vector& V,const Vector& W);
Vector operator*(const Vector& V,const Vector& W);
double operator^(const Vector& V,const Vector& W);

#endif


// Matrix.cpp

#include "Matrix.h"

// Multiplies a matrix by a vector
Vector operator*(const Matrix& C,const Vector& V)
{
   int d = C.size();
   Vector W(d);
   for (int j=0; j<d; j++)
   {
      W[j]=0.0;
      for (int l=0; l<d; l++) W[j]=W[j]+C[j][l]*V[l];
   }
   return W;
}

// Adds two vectors element-wise.
Vector operator+(const Vector& V,const Vector& W)
{
   int d = V.size();
   Vector U(d);
   for (int j=0; j<d; j++) U[j] = V[j] + W[j];
   return U;
}

// Adds a scalar to each element of a vector.
Vector operator+(const double& a,const Vector& V)
{
   int d = V.size();
   Vector U(d);
   for (int j=0; j<d; j++) U[j] = a + V[j];
```

```
   return U;
}

// Subtracts two vectors element-wise.
Vector operator-(const Vector& V,const Vector& W)
{
   int d = V.size();
   Vector U(d);
   for (int j=0; j<d; j++) U[j] = V[j] - W[j];
   return U;
}

// Subtracts a scalar to each element of a vector.
Vector operator-(const Vector& V, const double& a)
{
   int d = V.size();
   Vector U(d);
   for (int j=0; j<d; j++) U[j] = V[j] - a;
   return U;
}

// Multiplies each element of a vector by a scalar.
Vector operator*(const double& a,const Vector& V)
{
   int d = V.size();
   Vector U(d);
   for (int j=0; j<d; j++) U[j] = a*V[j];
   return U;
}

// Multiplies two vectors element-wise.
Vector operator*(const Vector& V,const Vector& W)
{
   int d = V.size();
   Vector U(d);
   for (int j=0; j<d; j++) U[j] = V[j] * W[j];
   return U;
}

// Computes the dot product of two vectors.
double operator^(const Vector& V,const Vector& W)
{
   double sum = 0.0;
   int d = V.size();
   for (int j=0; j<d; j++) sum = sum + V[j]*W[j];
   return sum;
}
```