# ICBS Computational Finance with C++

# Coursework

# Name: Ronald Leung

# CID: 02422598

## Software Structure

A diagram of functions and functions

Description automatically generated

Figure 1 Class diagram

There are 5 classes and 3 functions in this program, as illustrated in figure 1.

|  |  |
| --- | --- |
| **Class** | ParameterEstimation |
|  | * Calculate mean return and covariance matrix |
| **Class** | TargetReturns |
|  | * Create a vector of target return (from 0% to 10%) |
| **Class** | MatrixCaculator |
|  | * Implement basic matrix operations |
| **Sub Class** | ConjugateGradientMethod, a sub class of MatrixCalculator |
|  | * Conjugate Gradient Method |
| **Class** | Quadratic |
|  | * Prepare Q matrix, b matrix, initial guess x0 vector * Select a specific range of the return matrix * Calculate OOS returns |

|  |  |
| --- | --- |
| **Function** | readData |
|  | * Read CSV files to get stock returns, original given function |
| **Function** | String\_to\_double |
|  | * Convert string to doble |
| **Function** | MatrixWriter |
|  | * Write results to CSV files |

A screenshot of a computer program

Description automatically generated

Figure 2 main.cpp structure

Figure 2 illustrates the structure of main.cpp. The main.cpp can be divided into 4 sessions with a structure similar to the coursework instructions.

The main idea is to make C++ code in the main.cpp file feels like Python. With the classes and functions that are built previously in figure1, the modular code in main.cpp could be clean and easy to debug.

## Evaluation

A graph with many colored dots

Description automatically generated

Figure 3 Actual Return vs Variance for each iteration

Figure 3 plots the actual return across different level of variance. Each point represents a portfolio at a specific iteration with a given target return.

A diagram of a graph

Description automatically generated with medium confidence

Figure 4 Efficient Frontier vs OOS Actual Return, Expected Volatility (standard deviation) as x-axis

Figure 4 compares the in-sample efficient frontier with the out-of-sample performance across multiple iterations.

The blue curve represents the efficient frontier calculated using all in-sample data. This curve shows the optimal portfolios that provide the maximum expected return for a given level of risk (expected volatility) based on historical data.

Coloured points represent the out-of-sample performance of portfolios across different iterations. The points are scattered around the in-sample efficient frontier (blue curve), indicating variability in performance when the model is tested using out-of-sample data. The variability can be attributed to the limitation of the Markowitz model, changing market conditions, and model overfitting to the sub-set of data.

Even though the out-of-sample points are scattered around the efficient frontier, it is still able to observe the risk-return trade-off that is inherent in portfolio optimisation. The efficient frontier represents the theoretical best trade-off, while the out-of-sample points show the real-world deviations.

The scatter of out-of-sample points emphasizes the importance of robust back-testing. It shows that relying solely on in-sample data for portfolio optimization can be misleading. Continuous back-testing with rolling windows helps in assessing the stability and reliability of the optimization model, providing a more realistic expectation of future performance. A graph of a number of dots

Description automatically generated with medium confidenceA graph of a diagram

Description automatically generated with medium confidenceA graph of blue and pink dots

Description automatically generatedA graph of a graph

Description automatically generated with medium confidence

Figure 5 Out of sample actual returns vs target returns, compare with all iterations

The **3D scatter plot** shows the relationship between the iterations, target returns, and the difference between the target return and the actual return. Each point represents a portfolio at a specific iteration with a given target return, coloured by the magnitude of the difference.

From the **top view**, it indicates that the effectiveness of the portfolio optimization process varies over time (across iterations). It is suggested that market conditions could change from time to time, and it is important to periodically reassess and rebalance the portfolio to adapt to changing environments.

From the **side view**, it suggests that portfolios with higher target returns tend to underperform more compared to their targets. It highlights that aiming for higher returns using the Markowitz model might lead to larger deviations from the target. This could be due to an increased risk-return trade-off, a higher level of risk associated with a high target return can lead to greater deviations from the target.

The **Front view and Top view** show trends and cycles in the portfolio’s performance. Negative differences (target return – actual return) occur periodically with temporal patterns. There could be some time-varying factors that are impacting the portfolios’ performance which require further investigations.

## Appendix

C++ source code and python plots are zipped for your further reference.

|  |
| --- |
| Main.cpp |
| //g++ -std=c++17 -c main.cpp  //g++ -std=c++17 -c ParameterEstimation.cpp  //g++ -std=c++17 -c TargetReturns.cpp  //g++ -std=c++17 -c MatrixCalculator.cpp  //g++ -std=c++17 -c ConjugateGradientMethod.cpp  //g++ -std=c++17 -c csv.cpp  //g++ -std=c++17 -c string\_to\_double.cpp  //g++ -std=c++17 -c readData.cpp  //g++ -std=c++17 -c Quadratic.cpp  //g++ -std=c++17 -c MatrixWriter.cpp  //g++ -std=c++17 -c main.cpp MatrixWriter.cpp Quadratic.cpp readData.cpp string\_to\_double.cpp ParameterEstimation.cpp TargetReturns.cpp MatrixCalculator.cpp ConjugateGradientMethod.cpp csv.cpp  //g++ -std=c++17 -o portfolioSolver main.o MatrixWriter.o Quadratic.o readData.o csv.o string\_to\_double.o ParameterEstimation.o TargetReturns.o MatrixCalculator.o ConjugateGradientMethod.o  //./portfolioSolver  #include <stdio.h>  #include <fstream>  #include <stdlib.h>  #include <sstream>  #include <iostream>  #include <numeric>  #include <vector>  #include <cmath>  #include "csv.h"  #include "ParameterEstimation.h"  #include "TargetReturns.h"  #include "MatrixCalculator.h"  #include "ConjugateGradientMethod.h"  #include "string\_to\_double.h"  #include "readData.h"  #include "Quadratic.h"  #include "MatrixWriter.h"    typedef vector<vector<double>> Matrix; // define a type for a matrix  int main (int argc, char \*argv[])  {  int numberAssets=83;  int numberReturns=700;    TargetReturns target\_1(0.0, 0.1, 21);  vector<double> target\_returns = target\_1.getReturns(); // get the vector of target returns  double tolerance = 1e-10; // set the tolerance for the conjugate gradient method  Matrix returnMatrix(numberAssets, vector<double>(numberReturns, 0.0)); // create a matrix to store the returns of the assets  string fileName="asset\_returns.csv";  readData(returnMatrix,fileName); //read the data from the file and store it into the return matrix  Quadratic portfolio; // create a Quadratic object for portfolio optimization  vector<vector<vector<double>>> results;  vector<vector<vector<double>>> results\_2;    for (int i = 0; i < numberReturns - 100; i += 12) {  int index = i / 12;  int start = i;  int mid = i + 100;  int end = i + 112;    //print out the start, mid and end    cout<<"Processed window " << (start + 1) << " - " << (start + 100)<< " of "<<numberReturns<<endl;  Matrix returnMatrix\_IS = portfolio.selectColumns(returnMatrix, start, mid); // select the in-sample data  Matrix returnMatrix\_OOS = portfolio.selectColumns(returnMatrix, mid, end); // select the out-of-sample data    ParameterEstimation return\_IS(returnMatrix\_IS); // create a ParameterEstimation object for the in-sample data  vector<double> meanReturns\_IS = return\_IS.calculateMeanReturns(); // calculate the mean returns of the in-sample data  Matrix covarianceMatrix\_IS = return\_IS.calculateCovarianceMatrix(); // calculate the covariance matrix of the in-sample data  //Calculate the matrix Q, b\_matrix and x0 for ConjugateGradientMethod  Matrix Q = portfolio.createMatrixQ(covarianceMatrix\_IS, meanReturns\_IS); // create the Q matrix  Matrix b\_matrix = portfolio.calculateBMatrix(target\_returns, meanReturns\_IS); // create the b matrix  vector <double> x0 = portfolio.calculateX0(Q, numberAssets, 0.5); // calculate the initial guess for the weights  Matrix df\_weights = portfolio.calculatePortfolioWeights(Q, b\_matrix, x0, tolerance, meanReturns\_IS); // calculate the optimal weights  Matrix OOS\_returns = portfolio.backtesting(df\_weights, returnMatrix\_OOS, target\_returns); // backtesting the optimal weights  //store the results  results.push\_back(OOS\_returns);  results\_2.push\_back(df\_weights);  }    //write the results to a csv file  writeMatrixToCSV(results, "OOS\_returns.csv");  writeMatrixToCSV(results\_2, "df\_weights.csv");      return 0;  } |

|  |
| --- |
| ParameterEstimation.h |
| // ParameterEstimation.h  #ifndef PARAMETERESTIMATION\_H  #define PARAMETERESTIMATION\_H  #include <vector>  class ParameterEstimation {  private:  std::vector<std::vector<double>> returnMatrix;  public:  ParameterEstimation(const std::vector<std::vector<double>>& returnMatrix);  std::vector<double> calculateMeanReturns();  std::vector<std::vector<double>> calculateCovarianceMatrix();  };  #endif // PARAMETERESTIMATION\_H |

|  |
| --- |
| ParameterEstimation.cpp |
| // ParameterEstimation.cpp  #include "ParameterEstimation.h"  #include <numeric>  ParameterEstimation::ParameterEstimation(const std::vector<std::vector<double>>& returnMatrix) : returnMatrix(returnMatrix) {}  std::vector<double> ParameterEstimation::calculateMeanReturns() {  std::vector<double> meanReturns;  for (const auto& assetReturns : returnMatrix) {  double sum = std::accumulate(assetReturns.begin(), assetReturns.end(), 0.0);  meanReturns.push\_back(sum / assetReturns.size());  }  return meanReturns;  }  std::vector<std::vector<double>> ParameterEstimation::calculateCovarianceMatrix() {  std::vector<double> meanReturns = calculateMeanReturns();  int numAssets = returnMatrix.size();  int numReturns = returnMatrix[0].size();  std::vector<std::vector<double>> covarianceMatrix(numAssets, std::vector<double>(numAssets, 0.0));  for (int i = 0; i < numAssets; ++i) {  for (int j = 0; j < numAssets; ++j) {  double cov = 0.0;  for (int k = 0; k < numReturns; ++k) {  cov += (returnMatrix[i][k] - meanReturns[i]) \* (returnMatrix[j][k] - meanReturns[j]);  }  covarianceMatrix[i][j] = cov / (numReturns - 1);  }  }  return covarianceMatrix;  } |

|  |
| --- |
| TargetReturns.h |
| // TargetReturns.h  #ifndef TARGETRETURNS\_H  #define TARGETRETURNS\_H  #include <vector>  class TargetReturns {  private:  std::vector<double> target\_returns;  public:  TargetReturns(double start, double end, int num);  std::vector<double> getReturns() const;  };  #endif // TARGETRETURNS\_H |

|  |
| --- |
| TargetReturns.cpp |
| // TargetReturns.cpp  #include "TargetReturns.h"  TargetReturns::TargetReturns(double start, double end, int num) {  double step = (end - start) / (num - 1);  for (int i = 0; i < num; ++i) {  target\_returns.push\_back(start + i \* step);  }  }  std::vector<double> TargetReturns::getReturns() const {  return target\_returns;  } |

|  |
| --- |
| MatrixCalculator.h |
| // MatrixCalculator.h  #ifndef MATRIXCALCULATOR\_H  #define MATRIXCALCULATOR\_H  #include <vector>  class MatrixCalculator {  public:  double dotProduct(const std::vector<double>& a, const std::vector<double>& b);  std::vector<double> matrixVectorProduct(const std::vector<std::vector<double>>& a, const std::vector<double>& x);  std::vector<double> vectorSubtraction(const std::vector<double>& a, const std::vector<double>& b);  std::vector<double> vectorAddition(const std::vector<double>& a, const std::vector<double>& b);  std::vector<double> scalarVectorProduct(double scalar, const std::vector<double>& v);  };  #endif // MATRIXCALCULATOR\_H |

|  |
| --- |
| MatrixCalculator.cpp |
| // MatrixCalculator.cpp  #include "MatrixCalculator.h"  double MatrixCalculator::dotProduct(const std::vector<double>& a, const std::vector<double>& b) {  double sum = 0;  for (size\_t i = 0; i < a.size(); i++) {  sum += a[i] \* b[i];  }  return sum;  }  std::vector<double> MatrixCalculator::matrixVectorProduct(const std::vector<std::vector<double>>& a, const std::vector<double>& x) {  std::vector<double> result(a.size(), 0);  for (size\_t i = 0; i < a.size(); i++) {  for (size\_t j = 0; j < a[0].size(); j++) {  result[i] += a[i][j] \* x[j];  }  }  return result;  }  std::vector<double> MatrixCalculator::vectorSubtraction(const std::vector<double>& a, const std::vector<double>& b) {  std::vector<double> result(a.size());  for (size\_t i = 0; i < a.size(); i++) {  result[i] = a[i] - b[i];  }  return result;  }  std::vector<double> MatrixCalculator::vectorAddition(const std::vector<double>& a, const std::vector<double>& b) {  std::vector<double> result(a.size());  for (size\_t i = 0; i < a.size(); i++) {  result[i] = a[i] + b[i];  }  return result;  }  std::vector<double> MatrixCalculator::scalarVectorProduct(double scalar, const std::vector<double>& v) {  std::vector<double> result(v.size());  for (size\_t i = 0; i < v.size(); i++) {  result[i] = scalar \* v[i];  }  return result;  } |

|  |
| --- |
| ConjugateGradientMethod.h |
| // ConjugateGradientMethod.h  #ifndef CONJUGATEGRADIENTMETHOD\_H  #define CONJUGATEGRADIENTMETHOD\_H  #include <vector>  #include "MatrixCalculator.h"  class ConjugateGradientMethod : public MatrixCalculator {  public:  std::vector<double> conjugate\_gradient(const std::vector<std::vector<double>>& Q, const std::vector<double>& b, std::vector<double> x0, double tol=1e-6);  };  #endif // CONJUGATEGRADIENTMETHOD\_H |

|  |
| --- |
| ConjugateGradientMethod.cpp |
| // ConjugateGradientMethod.cpp  #include "ConjugateGradientMethod.h"  std::vector<double> ConjugateGradientMethod::conjugate\_gradient(const std::vector<std::vector<double>>& Q, const std::vector<double>& b, std::vector<double> x0, double tol) {  std::vector<double> s0 = vectorSubtraction(b, matrixVectorProduct(Q, x0));  std::vector<double> p0 = s0;  int i = 0;  while (dotProduct(s0, s0) > tol) {  double alpha = dotProduct(s0, s0) / dotProduct(matrixVectorProduct(Q, p0), p0);  x0 = vectorAddition(x0, scalarVectorProduct(alpha, p0));  std::vector<double> s1 = vectorSubtraction(s0, scalarVectorProduct(alpha, matrixVectorProduct(Q, p0)));  double beta = dotProduct(s1, s1) / dotProduct(s0, s0);  p0 = vectorAddition(s1, scalarVectorProduct(beta, p0));  s0 = s1;  i++;  }  return x0;  } |

|  |
| --- |
| csv.h |
| #ifndef \_CSV\_H  #define \_CSV\_H  #include <iostream>  #include <algorithm>  #include <string>  #include <vector>  using namespace std;  class Csv { // read and parse comma-separated values  // sample input: "LU",86.25,"11/4/1998","2:19PM",+4.0625    public:  Csv(istream& fin = cin, string sep = ",") :  fin(fin), fieldsep(sep) {}    int getline(string&);  string getfield(int n);  int getnfield() const { return nfield; }    private:  istream& fin; // input file pointer  string line; // input line  vector<string> field; // field strings  int nfield; // number of fields  string fieldsep; // separator characters    int split();  int endofline(char);  int advplain(const string& line, string& fld, int);  int advquoted(const string& line, string& fld, int);  };  #endif |

|  |
| --- |
| csv.cpp |
| #include "csv.h"  // endofline: check for and consume \r, \n, \r\n, or EOF  int Csv::endofline(char c)  {  int eol;    eol = (c=='\r' || c=='\n');  if (c == '\r') {  fin.get(c);  if (!fin.eof() && c != '\n')  fin.putback(c); // read too far  }  return eol;  }  // getline: get one line, grow as needed  int Csv::getline(string& str)  {  char c;    for (line = ""; fin.get(c) && !endofline(c); )  line += c;  split();  str = line;  return !fin.eof();  }  // split: split line into fields  int Csv::split()  {  string fld;  int i, j;    nfield = 0;  if (line.length() == 0)  return 0;  i = 0;    do {  if (i < line.length() && line[i] == '"')  j = advquoted(line, fld, ++i); // skip quote  else  j = advplain(line, fld, i);  if (nfield >= field.size())  field.push\_back(fld);  else  field[nfield] = fld;  nfield++;  i = j + 1;  } while (j < line.length());    return nfield;  }  // advquoted: quoted field; return index of next separator  int Csv::advquoted(const string& s, string& fld, int i)  {  int j;    fld = "";  for (j = i; j < s.length(); j++) {  if (s[j] == '"' && s[++j] != '"') {  int k = s.find\_first\_of(fieldsep, j);  if (k > s.length()) // no separator found  k = s.length();  for (k -= j; k-- > 0; )  fld += s[j++];  break;  }  fld += s[j];  }  return j;  }  // advplain: unquoted field; return index of next separator  int Csv::advplain(const string& s, string& fld, int i)  {  int j;    j = s.find\_first\_of(fieldsep, i); // look for separator  if (j > s.length()) // none found  j = s.length();  fld = string(s, i, j-i);  return j;  }  // getfield: return n-th field  string Csv::getfield(int n)  {  if (n < 0 || n >= nfield)  return "";  else  return field[n];  } |

|  |
| --- |
| string\_to\_double.h |
| #include <string>  #include <sstream>  double string\_to\_double(const std::string& s); |

|  |
| --- |
| string\_to\_double.cpp |
| #include "string\_to\_double.h"  double string\_to\_double(const std::string& s)  {  std::istringstream i(s);  double x;  if (!(i >> x))  return 0;  return x;  } |

|  |
| --- |
| readData.h |
| #include <vector>  #include <string>  void readData(std::vector<std::vector<double>>& data, const std::string& fileName); |

|  |
| --- |
| readData.cpp |
| #include "readData.h"  #include "string\_to\_double.h"  #include <fstream>  #include <iostream>  #include "csv.h"  void readData(std::vector<std::vector<double>>& data, const std::string& fileName)  {  std::ifstream file(fileName);  Csv csv(file);  std::string line;  if (file.is\_open())  {  int i = 0;  while (csv.getline(line) != 0) {  for (int j = 0; j < csv.getnfield(); j++)  {  double temp = string\_to\_double(csv.getfield(j));  //std::cout << "Asset " << j << ", Return " << i << "=" << temp << "\n";  data[j][i] = temp;  }  i++;  }  file.close();  }  else {  std::cout << fileName << " missing\n";  exit(0);  }  return;  } |

|  |
| --- |
| Quadratic.h |
| // Quadratic.h  #ifndef QUADRATIC\_H  #define QUADRATIC\_H  #include <vector>  class Quadratic {  public:  std::vector<std::vector<double>> createMatrixQ(const std::vector<std::vector<double>>& cov\_matrix, const std::vector<double>& mean\_returns);  std::vector<std::vector<double>> calculateBMatrix(const std::vector<double>& target\_returns, const std::vector<double>& meanReturns\_small);  std::vector<double> calculateX0(const std::vector<std::vector<double>>& Q, int numberAssets, double initial\_guess = 0.5);  std::vector<std::vector<double>> calculatePortfolioWeights(const std::vector<std::vector<double>>& Q, const std::vector<std::vector<double>>& b\_matrix, const std::vector<double>& x0, double tolerance, const std::vector<double>& meanReturns\_small);  std::vector<std::vector<double>> createSmallReturnMatrix(const std::vector<std::vector<double>>& returnMatrix, int numberAssets\_small, int numberReturns\_small);  std::vector<std::vector<double>> selectRows(const std::vector<std::vector<double>>& returnMatrix, int start, int end);  std::vector<std::vector<double>> selectColumns(const std::vector<std::vector<double>>& returnMatrix, int start, int end);  std::vector<std::vector<double>> backtesting(const std::vector<std::vector<double>>& optimal\_weights, const std::vector<std::vector<double>>& OOS\_returns, const std::vector<double>& target\_return);  };  #endif |

|  |
| --- |
| Quadratic.cpp |
| // Quadratic.cpp  #include "Quadratic.h"  #include "ConjugateGradientMethod.h"  #include "ParameterEstimation.h"  #include "MatrixCalculator.h"  #include <iostream>  std::vector<std::vector<double>> Quadratic::createMatrixQ(const std::vector<std::vector<double>>& cov\_matrix, const std::vector<double>& mean\_returns) {  int len = mean\_returns.size();  std::vector<std::vector<double>> Q(len + 2, std::vector<double>(len + 2, 0));  // Copy cov\_matrix into Q  for (int i = 0; i < len; i++) {  for (int j = 0; j < len; j++) {  Q[i][j] = cov\_matrix[i][j];  }  }  // Copy mean\_returns and -1 into Q  for (int i = 0; i < len; i++) {  Q[len][i] = -mean\_returns[i];  Q[len + 1][i] = -1;  }  // Copy array\_ and array\_2 into Q  for (int i = 0; i < len; i++) {  Q[i][len] = -mean\_returns[i];  Q[i][len + 1] = -1;  }  /\*  // Print Q  std::cout << "\n\_\_\_\_\_\_\_\n Matrix Q:\n";  for (const auto& row : Q) {  for (const auto& elem : row) {  std::cout << elem << " ";  }  std::cout << "\n";  }  std::cout << "(" << Q.size() << ", " << Q[0].size() << ")\n";  \*/  return Q;  }  std::vector<std::vector<double>> Quadratic::calculateBMatrix(const std::vector<double>& target\_returns, const std::vector<double>& meanReturns\_small) {  int n = target\_returns.size();  int mean\_returns\_len = meanReturns\_small.size();  std::vector<std::vector<double>> b\_matrix(n, std::vector<double>(mean\_returns\_len + 2, 0.0));  for (int i = 0; i < n; ++i) {  b\_matrix[i][mean\_returns\_len] = -target\_returns[i];  b\_matrix[i][mean\_returns\_len + 1] = -1.0;  }  /\*    //print out b\_matrix  std::cout << "\n\_\_\_\_\_\_\_\nPrinting out b\_matrix\n";  for(int i = 0; i < n; i++) {  for(int j = 0; j < mean\_returns\_len + 2; j++) {  std::cout << b\_matrix[i][j] << " ";  }  std::cout << "\n";  }  \*/  return b\_matrix;  }  std::vector<double> Quadratic::calculateX0(const std::vector<std::vector<double>>& Q, int numberAssets, double initial\_guess) {  int x0\_size = Q.size();  double fillValue = initial\_guess;  std::vector<double> x0(x0\_size, fillValue);  /\*    //print out x0  std::cout << "\n\_\_\_\_\_\_\_\nPrinting out x0 in the main function\n";  for(int i = 0; i < numberAssets + 2; i++) {  std::cout << x0[i] << " ";  }  \*/  return x0;  }  std::vector<std::vector<double>> Quadratic::calculatePortfolioWeights(const std::vector<std::vector<double>>& Q, const std::vector<std::vector<double>>& b\_matrix, const std::vector<double>& x0, double tolerance, const std::vector<double>& meanReturns\_small) {  std::vector<std::vector<double>> weights\_list;  ConjugateGradientMethod solver;  for (int i = 0; i < b\_matrix.size(); ++i) {  std::vector<double> weight = solver.conjugate\_gradient(Q, b\_matrix[i], x0, tolerance);  weight.resize(meanReturns\_small.size());  weights\_list.push\_back(weight);  }  /\*    std::cout << "\nWeights Matrix: \n";  for (const auto& weights : weights\_list) {  for (const auto& x : weights) {  std::cout << x << " ";  }  std::cout << "\n";  }  \*/  return weights\_list;  }  std::vector<std::vector<double>> Quadratic::createSmallReturnMatrix(const std::vector<std::vector<double>>& returnMatrix, int numberAssets\_small, int numberReturns\_small) {  std::vector<std::vector<double>> returnMatrix\_small(numberAssets\_small, std::vector<double>(numberReturns\_small, 0.0));  for(int i = 0; i < numberAssets\_small; i++) {  for(int j = 0; j < numberReturns\_small; j++) {  returnMatrix\_small[i][j] = returnMatrix[i][j];  }  }  return returnMatrix\_small;  }  std::vector<std::vector<double>> Quadratic::selectRows(const std::vector<std::vector<double>>& returnMatrix, int start, int end) {  //std::cout << "start: " << start << " end: " << end << std::endl;  std::vector<std::vector<double>> subset;  for (int i = start; i < end; ++i) {  subset.push\_back(returnMatrix[i]);    }  return subset;  }  std::vector<std::vector<double>> Quadratic::selectColumns(const std::vector<std::vector<double>>& returnMatrix, int start, int end) {  //std::cout << "start: " << start << " end: " << end << std::endl;  std::vector<std::vector<double>> subset;  for (const auto& row : returnMatrix) {  if (end > row.size()) {  end = row.size(); // Adjust end to be the size of row  }  std::vector<double> new\_row(row.begin() + start, row.begin() + end);  subset.push\_back(new\_row);  }  return subset;  }  std::vector<std::vector<double>> Quadratic::backtesting(const std::vector<std::vector<double>>& optimal\_weights, const std::vector<std::vector<double>>& OOS\_returns, const std::vector<double>& target\_return) {  ParameterEstimation return\_OOS(OOS\_returns);  std::vector<double> mean\_returns\_OOS = return\_OOS.calculateMeanReturns();  std::vector<std::vector<double>> covariance\_martix\_OOS = return\_OOS.calculateCovarianceMatrix();  MatrixCalculator calc;  std::vector<std::vector<double>> res;  for (int index = 0; index < optimal\_weights.size(); ++index) {  std::vector<double> weights = optimal\_weights[index];  double targ\_ret = target\_return[index];  double act\_ave\_return = calc.dotProduct(mean\_returns\_OOS, weights);  std::vector<double> temp = calc.matrixVectorProduct(covariance\_martix\_OOS, weights);  double pf\_cov = calc.dotProduct(weights, temp);  res.push\_back({targ\_ret, act\_ave\_return, pf\_cov});  }  return res;  } |

|  |
| --- |
| MatrixWriter.h |
| // In a header file called "MatrixWriter.h"  #ifndef MATRIX\_WRITER\_H  #define MATRIX\_WRITER\_H  #include <vector>  #include <string>  void writeMatrixToCSV(const std::vector<std::vector<std::vector<double>>>& matrix, const std::string& filename);  #endif |

|  |
| --- |
| MatrixWriter.cpp |
| // In a cpp file called "MatrixWriter.cpp"  #include "MatrixWriter.h"  #include <iostream>  #include <fstream>  void writeMatrixToCSV(const std::vector<std::vector<std::vector<double>>>& matrix, const std::string& filename) {  std::ofstream file(filename);  if (!file.is\_open()) {  std::cout << "Failed to open the file." << std::endl;  return;  }  for (const auto& subMatrix : matrix) {  for (const auto& row : subMatrix) {  for (size\_t i = 0; i < row.size(); ++i) {  file << row[i];  if (i != row.size() - 1) {  file << ",";  }  }  file << "\n";  }  file << "\n";  }  file.close();  } |