Computational Finance with C++

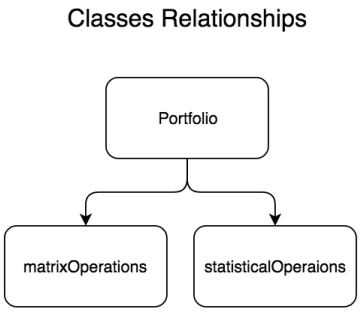
Individual Coursework

CID: 01267314

King Hang Wong

Software Structure:

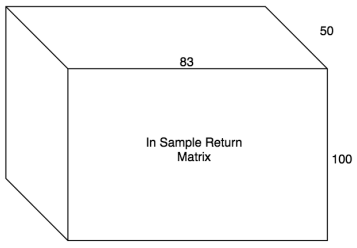
In this project, I created three different classes, including Portfolio, matrixOperations and statisticalOperations. The diagram below shows the relationship between 3 classes

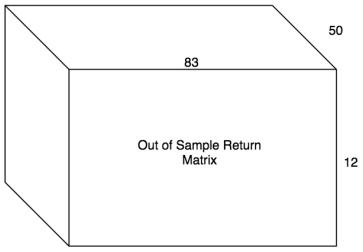


Therefore, the portfolio class will only call the functions inside matrixOperations and statisticalOperations, but not the other way round. For example, inside the optimization process, we called the static matrix multiplication function inside matrixOpertaion class in order to perform the matrix multiplication.

Processes:

At the beginning of the project, I started reading the return data by using the code provided. It returns a 2 dimensional array of 700 days return with 83 companies. Then I transform this array into a 2D vector in order to make the code consistent by using vectors. After that, I created in sample and out of sample matrices, which are both 3D vectors. These two matrices look like the following:



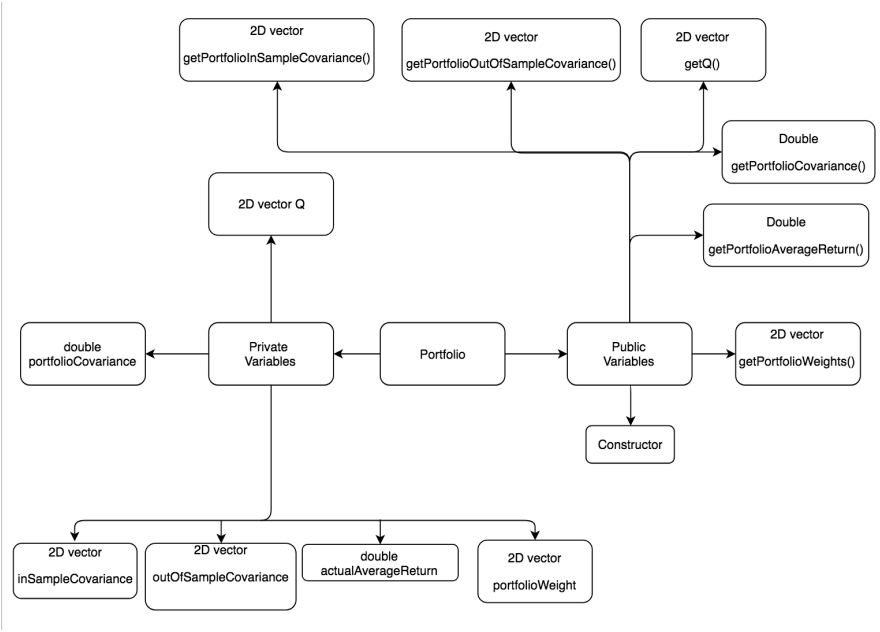


These two matrices are constructed by using two different functions, which are called inSampleRollingWindow and outOfSampleRollingWindow. These two functions will read in the 2D full sample return matrix and divide them into 50 2D in sample return matrices and 49 2D out of sample matrices. Then I created a 2D matrix matrixOfCompanyMeanReturn, which stores the mean return of the

companies in each of the rolling windows.

Then I created 3 different classes, including Portfolio, matrixOperations and statisticalOperations. Portfolio calls the other 2 classes, but they are 3 different classes, not inheritances. For both matrixoOperations and statisticalOperations, all the public variables inside these two classes are static, therefore we can share its functionality within all of the objects. These two classes include all the calculations require for this project.

Then I started to construct portfolios with different returns and different time periods. In total, I created 21 portfolios (from 0% to 10% with an increment of 0.5%) for 50 different periods, with the product of 1050 portfolios. And the diagram below can show the structures and variables of the portfolio.



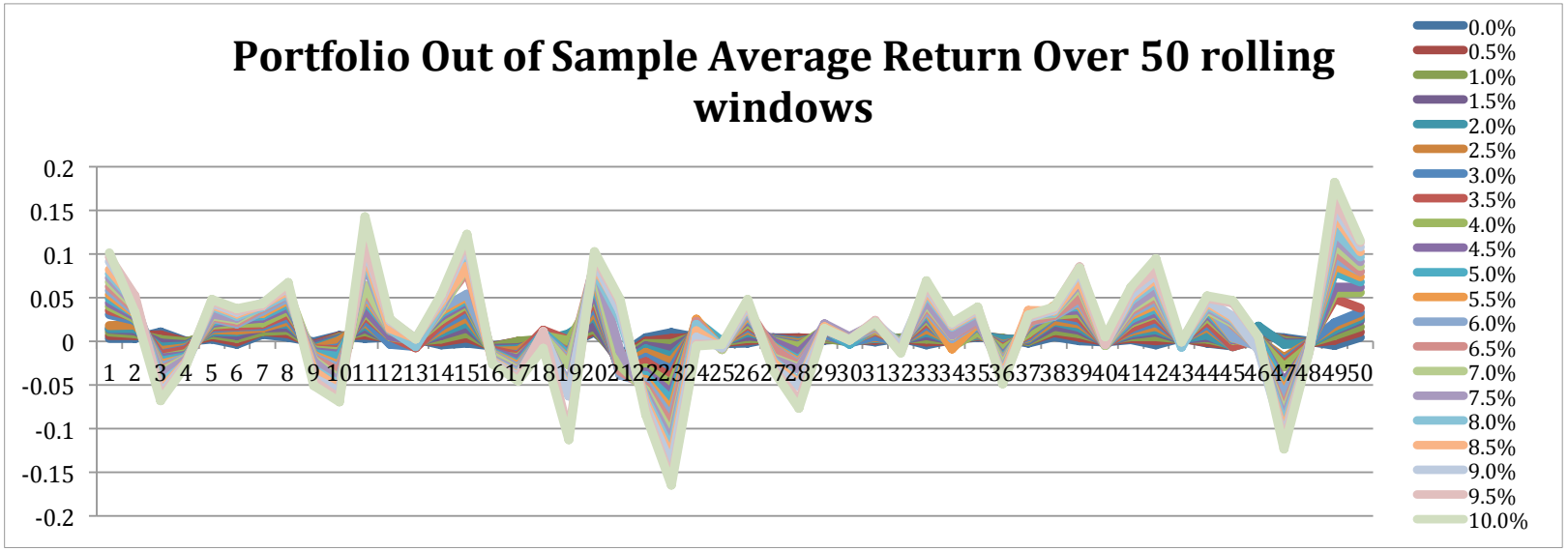
Then I use the call functions to obtain the out of sample portfolio average return and covariance and save them into 2D matrices (oosAverageReturn and oosCovariance respectively). Therefore these two matrices are our final results for this project and next part will be evaluation of our results.

Evaluation

The table below shows the average out of sample return from different portfolio targets (i.e. mean return over 50 out of sample average returns):

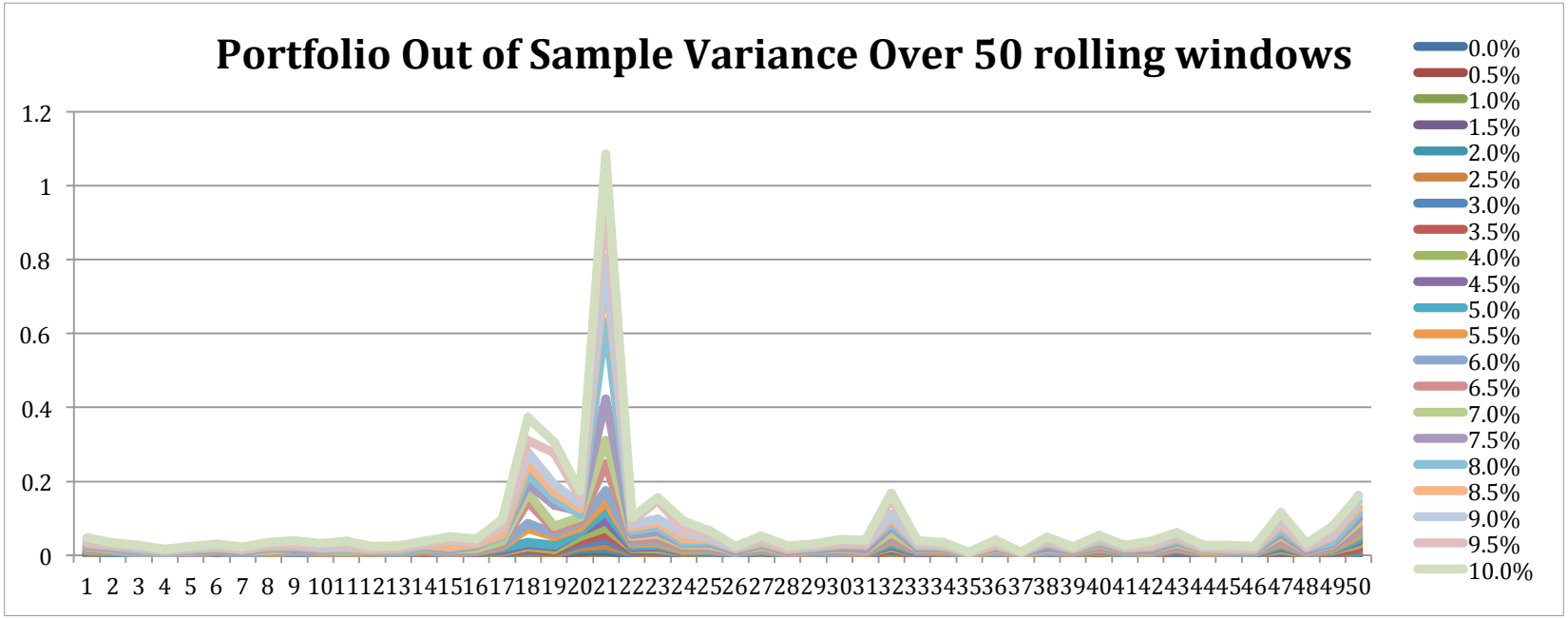
Target Return	**0%**	**0.50%**	**1%**	**1.50%**	**2%**	**2.50%**	**3%**	**3.50%**	**4%**	**4.50%**	**5%**
Average Return	0.1330%	0.2391%	0.3510%	0.4466%	0.5199%	0.5211%	0.5993%	0.7827%	0.8701%	0.9142%	1.0195%
Target Return	**5.50%**	**6%**	**6.50%**	**7%**	**7.50%**	**8%**	**8.50%**	**9%**	**9.50%**	**10%**	
Average Return	1.0368%	1.0733%	1.1956%	1.2118%	1.2916%	1.3734%	1.4522%	1.5314%	1.5223%	1.5820%	

As we can see in on the table above, as the target return increases, the out of sample average return will also increase. And the diagram below shows the changes of out of sample returns in different periods, we can see that the portfolio return fluctuates over the 50 periods of rolling windows, where the largest loss is about 16% for 10% target return.



The table below shows the standard deviation of the out of sample average return of the 21 portfolios:

Target Return	**0.00%**	**0.50%**	**1.00%**	**1.50%**	**2.00%**	**2.50%**	**3.00%**	**3.50%**	**4.00%**	**4.50%**	**5.00%**
Standard Deviation	0.4768%	0.5113%	0.6308%	0.8542%	1.1048%	1.3603%	1.6758%	1.9948%	2.3172%	2.6911%	3.0628%
Target Return	**5.50%**	**6.00%**	**6.50%**	**7.00%**	**7.50%**	**8.00%**	**8.50%**	**9.00%**	**9.50%**	**10.00%**	
Standard Deviation	3.4089%	3.6533%	4.0247%	4.3110%	4.6953%	5.0432%	5.3556%	5.7711%	6.3423%	6.8094%	



The diagram above shows the portfolio out of sample covariance (variance) over different rolling windows. We can see that the covariance increases when the loss of the portfolio increases and vice versa. And also we can see that the higher the target return, out of sample portfolio variance can also be higher.

In my opinion, the Markowitz model does not perform very well. As we can see, even we set the target return to be 10%, but the average return for all out of sample periods is about 1.5%. Therefore I think we can consider alternative models to optimize the return.

Note. Csv files for portfolio returns and portfolio variances can be found by running the code, and also attached with the zip file. And please run the following command for final result.

```
g++ -o portfolio mainwo.cpp portfolio.cp csv.cp matrixOperations.cpp statisticalOperations.cpp
```

Appendix:

mainwo.cpp

```
#include <cmath>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>

//g++ -o portfolio mainwo.cpp portfolio.cp csv.cp
matrixOperations.cpp statisticalOperations.cpp


#include "portfolio.h"
#include "statisticalOperations.h"
#include "csv.h"

using namespace std;

double string_to_double( const string& s );
void readData(double **data,string fileName);
vector<vector<vector<double> > > inSampleRollingWindow (int
inSampleRollingWindowSize, int outOfSampleRollingWindowSize, int
numberOfAssets, int numberOfDays, vector<vector<double> >
returnVector);
vector<vector<vector<double> > > outOfSampleRollingWindow (int
inSampleRollingWindowSize, int outOfSampleRollingWindowSize, int
numberOfAssets, int numberOfDays, vector<vector<double> >
returnVector);


int main()
{
   // declaring all the variables that I need to use
  int numberOfAssets = 83;
  int numberOfDays = 700;
```

```
  int inSampleRollingWindowSize = 100;
  int outOfSampleRollingWindowSize = 12;
  int numberOfRollingWindows = 50;
  int numberOfPortfolioReturns = 21;

  double **returnMatrix = new double*[numberOfAssets]; //matrix to
store the return data by allocating memroy for return data
  for (int i =0; i< numberOfAssets; i++)
    returnMatrix[i] = new double[numberOfDays];
  string fileName = "asset_returns.csv";
  readData(returnMatrix,fileName);

  vector< vector<double> > returnVector (numberOfAssets,
vector<double>(numberOfDays));

    // transforming array to vector
  for (int i = 0; i < numberOfAssets; i++)
  {
     for (int j = 0; j < numberOfDays; j++)
    {
     returnVector[i][j] = returnMatrix[i][j];
   }
   }
    // construction of 3D vectors for in sample rolling windows and
out of sample rolling windows
  vector<vector<vector<double> > > inSampleReturn =
inSampleRollingWindow (inSampleRollingWindowSize,
outOfSampleRollingWindowSize, numberOfAssets, numberOfDays,
returnVector);
  vector<vector<vector<double> > > outOfSampleReturn =
outOfSampleRollingWindow (inSampleRollingWindowSize,
outOfSampleRollingWindowSize, numberOfAssets, numberOfDays,
returnVector);
    // creating 2D vector for each company mean return over 50 rolling
windows
    // 1D vector for pushing into 2D vector
  vector<double> VectorOfcompanyMeanRet (numberOfAssets);
  vector<vector<double> > matrixOfCompanyMeanReturn;

  for  (int j = 0; j < numberOfRollingWindows; j++)
  {
```

```
    for (int i = 0 ; i < numberOfAssets; i++)
    {
       VectorOfcompanyMeanRet[i] =
StatisticalOperations::mean(inSampleReturn[j][i]);
    }
    matrixOfCompanyMeanReturn.push_back(VectorOfcompanyMeanRet);
  }
    // creating 2D vectors for saving all portfolios out of sample
portfolio returns and out of sample portfolio variance
  vector<vector<double> > oosAverageReturn
(numberOfPortfolioReturns,
vector<double>(numberOfRollingWindows));
  vector<vector<double> > oosCovariance (numberOfPortfolioReturns,
vector<double>(numberOfRollingWindows));
    // setting target return to be 0 to initialise different
portfolio's target returns (in total 21 portfolios)
  double targetReturn = 0.0;
    //loop through number of portfolio returns, which is 21
  for (int j = 0; j < numberOfPortfolioReturns; j++)
  { //loop through number of portfolio rolling windows, which is 50
    for (int i = 0; i < numberOfRollingWindows; i++)
    {
       //constructing different portfolios
     Portfolio portfolio(inSampleReturn[i],
matrixOfCompanyMeanReturn[i], numberOfAssets,
inSampleRollingWindowSize, numberOfDays,
outOfSampleRollingWindowSize, targetReturn, outOfSampleReturn[i]);
       // getting different return and variance from different
portfolios
     oosAverageReturn[j][i] =
portfolio.getPortfolioAverageReturn();
     oosCovariance[j][i] = portfolio.getPortfolioCovariance();
    }
    targetReturn += 0.005; //increment by 0.5% each time to create
21 portfolios
  }

    // output of csv files (these two files contain information for
portfolio return and portfolio variance

  ofstream myfile2;
```

```
   myfile2.open ("oosAverageReturn.csv");
   for (int j = 0 ; j < numberOfPortfolioReturns; j++)
      {for(int i = 0 ; i < numberOfRollingWindows; i++)
         {myfile2 << oosAverageReturn[j][i] << ",";}
         myfile2 << "\n";
      }
   myfile2.close();

  ofstream myfile1;
   myfile1.open ("oosCovariance.csv");
   for (int j = 0 ; j < numberOfPortfolioReturns; j++)
      {for(int i = 0 ; i < numberOfRollingWindows; i++)
         {myfile1 << oosCovariance[j][i] << ",";}
         myfile1 << "\n";
      }
   myfile1.close();


}
// these are codes, which are provided by lecturer
double string_to_double( const string& s )
{
    istringstream i(s);
    double x;
    if (!(i >> x))
      return 0;
    return x;
}

void readData(double **data,string fileName)
{
    char tmp[20];
    ifstream file (strcpy(tmp, fileName.c_str()));
    Csv csv(file);
    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));
            //cout << "Asset " << j << ", Return "<<i<<"="<<
temp<<"\n";
            data[j][i]=temp;
          }
          i++;
      }
      file.close();
    }
    else
    {
      cout <<fileName <<" missing\n";exit(0);
    }
}

// in sample rolling window function (3D matrix)
vector<vector<vector<double> > > inSampleRollingWindow (int
inSampleRollingWindowSize, int outOfSampleRollingWindowSize, int
numberOfAssets, int numberOfDays, vector<vector<double> >
returnVector)
{
  vector<vector<vector<double> > > tempBacktest;
  //(50, vector<vector<double> >(numberOfAssets,
vector<double>(inSampleRollingWindowSize)));
  vector<vector<double> > tempReturnVector (numberOfAssets,
vector<double> (inSampleRollingWindowSize));
    for (int j = 0; j < numberOfDays - inSampleRollingWindowSize; j
+= 12)
    {
     for (int k = 0; k < numberOfAssets; k++)
     {
       for (int i = 0; i < 100; i++)
       {
          tempReturnVector[k][i] = returnVector[k][(i+j)];
       }
     }
    tempBacktest.push_back(tempReturnVector);
    }
  return tempBacktest;
}
```

```
// out of sample rolling window function (3D matrix)
vector<vector<vector<double> > > outOfSampleRollingWindow (int
inSampleRollingWindowSize, int outOfSampleRollingWindowSize, int
numberOfAssets, int numberOfDays, vector<vector<double> >
returnVector)
{
  vector<vector<vector<double> > > tempBacktest;
  vector<vector<double> > tempReturnVector (numberOfAssets,
vector<double> (outOfSampleRollingWindowSize));
  for (int j = 100; j < numberOfDays; j += 12)
  {
    for (int k = 0; k < numberOfAssets; k++)
    {
      for (int i = 0; i < outOfSampleRollingWindowSize; i++)
      {
        tempReturnVector[k][i] = returnVector[k][(i+j)];
      }
    }
    tempBacktest.push_back(tempReturnVector);
  }

  return tempBacktest;
}
```

matrixOperations.cpp

```
#include <cmath>
#include <vector>

#include "matrixOperations.h"

using namespace std;

vector< vector<double> > MatrixOperations::plus(vector<
vector<double> > matrix1, vector< vector<double> > matrix2)
{
     int width = matrix1.size();
     int height = matrix1[0].size();
     vector<vector<double> > result(width, vector<double>(height));
    for (int i = 0; i < width; i++)
    {
        for (int j = 0; j < height; j++)
        {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }
    return result;
}

vector< vector<double> > MatrixOperations::minus(vector<
vector<double> > matrix1, vector< vector<double> > matrix2)
{
     int width = matrix1.size();
     int height = matrix1[0].size();
     vector<vector<double> > result(width, vector<double>(height));
    for (int i = 0; i < width; i++)
    {
        for (int j = 0; j < height; j++)
        {
            result[i][j] = matrix1[i][j] - matrix2[i][j];
        }
    }
    return result;
}
```

```
// this is a function to multiply two matrices
vector< vector<double> > MatrixOperations::multiple(vector<
vector<double> > matrix1, vector< vector<double> > matrix2)
{
    vector< vector <double> > result(matrix2.size(),
vector<double>(matrix1[0].size()));

   // Multiplying matrix a and b and storing in array mult.
   for (int i = 0; i < matrix1[0].size(); i++)
   {
      for (int j = 0; j < matrix2.size(); j++)
      {
         for (int k = 0; k < matrix1.size(); k++)
         {
            result[j][i] += matrix1[k][i] * matrix2[j][k];
         }
      }
   }
   return result;
}

// this is a function to multiply a scalar with a matrix
vector< vector<double> > MatrixOperations::scalarMultiple(double
scalar, vector< vector<double> > matrix)
{
    int width = matrix.size();
    int height = matrix[0].size();
    vector<vector<double> > result(width, vector<double>(height));
   for (int i = 0; i < width; i++)
   {
      for (int j = 0; j < height; j++)
      {
         result[i][j] = scalar * matrix[i][j];
      }
   }
   return result;
}

vector< vector<double> > MatrixOperations::transpose(vector<
vector<double> > matrix)
{
```

```
    int width = matrix.size();
    int height = matrix[0].size();
    vector<vector<double> > result(height, vector<double>(width));
   for (int i = 0; i < width; i++)
   {
      for (int j = 0; j < height; j++)
      {
         result[j][i] = matrix[i][j];
      }
   }
   return result;
}
```

matrixOperations.h

```
#ifndef MatrixOperations_h
#define MatrixOperations_h

using namespace std;

class MatrixOperations
{
    public:
      static vector<double> plus(vector<double> matrix1,
vector<double> matrix2);
      static vector< vector<double> > plus(vector< vector<double> >
matrix1, vector< vector<double> > matrix2);
      static vector<double> minus(vector<double> matrix1,
vector<double> matrix2);
      static vector< vector<double> > minus(vector< vector<double> >
matrix1, vector< vector<double> > matrix2);
      static vector< vector<double> > multiple(vector<
vector<double> > matrix1, vector< vector<double> > matrix2);
      static vector< vector<double> > scalarMultiple(double scalar,
vector< vector<double> > matrix);
      static vector< vector<double> > transpose(vector<
vector<double> > matrix);
};

#endif
```

portfolio.cp

```
#include <cmath>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <fstream>
#include <sstream>
#include <string>
#include <numeric>

#include "portfolio.h"
#include "csv.h"
#include "matrixOperations.h"
#include "statisticalOperations.h"

using namespace std;
// construction of portfolio
Portfolio::Portfolio(vector< vector<double> > inSampleMat,
vector<double> vectorOfCompanyMeanRet, int noOfCompany, int
inSampleRollingWindowSize, int numberOfDays, int
outOfSampleRollingWindowSize, double noOfTargetReturn,
vector<vector<double > > outOfSampleReturn)
{

   outOfSampleAverageReturn.resize((1),vector<double> (83));
   vector<vector<double> > tempNegativeRet (1, vector<double>
(83));

   for (int i = 0; i < 83; i++)
   {
      tempNegativeRet[0][i] = -1 * vectorOfCompanyMeanRet[i];
   }

   for (int k = 0; k < 83; k++)
   {
      for (int i = 0; i < outOfSampleRollingWindowSize; i++)
      {
         outOfSampleAverageReturn[0][k] =
StatisticalOperations::mean(outOfSampleReturn[k]);
      }
```

```
    }
    // creating in sample covariance by calling get covariance function
from Statistical Operations class (static)
    inSampleCovariance =
StatisticalOperations::getCovariance(inSampleMat, noOfCompany,
inSampleRollingWindowSize);
    // creating out of sample covariance by calling get covariance
function from Statistical Operations class (static)
    outOfSampleCovariance =
StatisticalOperations::getCovariance(outOfSampleReturn,
noOfCompany, outOfSampleRollingWindowSize);

    // creating Q matrix
    Q.resize((85),vector<double> (85));

    for (int j = 0; j < noOfCompany; j++)
    {
        for(int k = 0; k < noOfCompany ; k++)
        {
            Q[j][k] = inSampleCovariance[j][k];
        }
    }

    for (int j = 0; j < noOfCompany + 2; j++)
    {
        Q[j][83] = tempNegativeRet[0][j];
        Q[j][84] = -1;
        Q[83][j] = tempNegativeRet[0][j];
        Q[84][j] = -1;
    }

    Q[83][83] = 0;
    Q[83][84] = 0;
    Q[84][83] = 0;
    Q[84][84] = 0;

    //creating temp portfolio weight vector
    vector <double> tempPortfolioWeight(noOfCompany);
    // getWeights function returns the weights after optimisation
(Conjugate Gradient Method)
```

```
    tempPortfolioWeight = StatisticalOperations::getWeights(Q,
noOfCompany, noOfTargetReturn);
    // transforming it into vector of vector since all of the matrix
operations are in vector of vector form (i.e. 1x83 or 83x1)
    vector <vector<double> > portfolioWeights;
    portfolioWeights.push_back(tempPortfolioWeight);

    // return portfolio varaince and portfolio return
    portfolioCovariance =
MatrixOperations::multiple(MatrixOperations::transpose(portfolioW
eights),
MatrixOperations::multiple(outOfSampleCovariance,portfolioWeights
))[0][0];
    actualAverageReturn =
MatrixOperations::multiple(MatrixOperations::transpose(outOfSampl
eAverageReturn),portfolioWeights)[0][0];
};

// all get functions are declared here
vector<vector<double> > Portfolio::getPortfolioWeights()
{
    return portfolioWeight;
};

vector<vector<double> >
Portfolio::getPortfolioInSampleCovariance()
{
    return inSampleCovariance;
};

vector<vector<double> > Portfolio::getQ()
{
    return Q;
}

vector<vector<double> >
Portfolio::getPortfolioOutOfSampleCovariance()
{
    return outOfSampleCovariance;
}
```

```
double Portfolio::getPortfolioCovariance()
{
    return portfolioCovariance;
}

double Portfolio::getPortfolioAverageReturn()
{
    return actualAverageReturn;
}
```

portfolio.h

```
#ifndef Portfolio_h
#define Portfolio_h

#include <cmath>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>

using namespace std;

class Portfolio
{
    private:
      vector< vector<double> > inSampleCovariance;
      vector< vector<double> > outOfSampleCovariance;
      vector< vector<double> > outOfSampleAverageReturn;
      double actualAverageReturn;
      vector<vector<double > > portfolioWeight;
      vector< vector<double> > Q;
      double portfolioCovariance;

    public:
      Portfolio(vector< vector<double> > inSampleReturn, vector<
double > matrixOfCompanyMeanRet, int noOfCompany, int
inSampleRollingWindowSize, int numberOfDays, int
outOfSampleRollingWindowSize, double noOfTargetReturn,
vector<vector<double> > outOfSampleReturn);
      vector<vector<double> > getPortfolioWeights();
      vector<vector<double> > getPortfolioInSampleCovariance();
      vector<vector<double> > getPortfolioOutOfSampleCovariance();
      vector<vector<double> > getQ();
      double getPortfolioCovariance();
      double getPortfolioAverageReturn();
};
#endif
```

statisticalOperations.cpp

```cpp
#include <cmath>
#include <vector>

#include "statisticalOperations.h"
#include "matrixOperations.h"

using namespace std;

double StatisticalOperations::mean(vector<double> input)
{
    double sum = 0.0;
    for (int i = 0; i < input.size(); i++)
    {
        sum += input[i];
    }
    return (sum / input.size());
}

double StatisticalOperations::standardDeviation(vector<double>
input , double mean)
{
    double sumSQ = 0.0;
    for (int i = 0; i < input.size(); i++)
    {
        sumSQ += (input[i] - mean) * (input[i] - mean);
    }
    return (sqrt(sumSQ / (input.size() - 1 )));
}

vector< vector<double> >
StatisticalOperations::getCovariance(vector< vector<double> >
returnVector, int numberOfCompany, int timeLength)
{
    vector< vector<double> > cov(numberOfCompany,
vector<double>(numberOfCompany));

    vector<double> firstCompany(timeLength);
    vector<double> secondCompany(timeLength);
```

```
    for (int i = 0; i < numberOfCompany; i++)
    {
        for (int k = 0; k < numberOfCompany; k++)
        {
            for (int j = 0; j < timeLength; j++)
            {
                firstCompany[j] = returnVector[i][j];
                secondCompany[j] = returnVector[k][j];
            }

            double firstCompanyMean = mean(firstCompany);
            double secondCompanyMean = mean(secondCompany);

            for (int j = 0; j < timeLength; j++)
            {
                cov[i][k] += (firstCompany[j] - firstCompanyMean) *
(secondCompany[j] - secondCompanyMean) / (timeLength - 1);
            }
        }
    }
    return cov;
}

// this includes conjugate gradient method

vector<double> StatisticalOperations::getWeights(vector<
vector<double> > Q, double numberOfCompany, double noOfTargetReturn)
{
    double tolerence = 0.000001;

    // Set up x
    vector< vector<double> > x(1, vector<double>(numberOfCompany +
2));
    for (int i = 0; i < numberOfCompany; i++)
    {
        x[0][i] = 1.0 / numberOfCompany;
    }
    x[0][numberOfCompany] = 1.0; // lambda
    x[0][numberOfCompany + 1] = 1.0; // mu

    // Set up b
```

```
    vector< vector<double> > b(1, vector<double>(numberOfCompany +
2));
    for (int i = 0; i < numberOfCompany; i++)
    {
        b[0][i] = 0.0;
    }
    b[0][numberOfCompany] = -1.0 * noOfTargetReturn; // -r_p
    b[0][numberOfCompany + 1] = -1.0;

    vector< vector<double> > s = MatrixOperations::minus(b,
MatrixOperations::multiple(Q, x));
    vector< vector<double> > p(s);

    double sTs =
MatrixOperations::multiple(MatrixOperations::transpose(s),
s)[0][0];
    while (sTs > tolerence)
    {
        double alpha = sTs /
(MatrixOperations::multiple(MatrixOperations::multiple(MatrixOper
ations::transpose(p), Q), p)[0][0]);

        x = MatrixOperations::plus(x,
(MatrixOperations::scalarMultiple(alpha, p)));

        vector< vector<double> > s_plus1 = MatrixOperations::minus(s,
(MatrixOperations::scalarMultiple(alpha,
(MatrixOperations::multiple(Q, p)))));

        sTs =
MatrixOperations::multiple(MatrixOperations::transpose(s_plus1),
s_plus1)[0][0];

        double beta = (sTs) /
(MatrixOperations::multiple(MatrixOperations::transpose(s),
s)[0][0]);

        p = MatrixOperations::plus(s_plus1,
(MatrixOperations::scalarMultiple(beta, p)));

        s = s_plus1;
```

```
    }

    vector<double> weights (numberOfCompany);
    for (int i = 0; i < weights.size(); i++)
    {
        weights[i] = x[0][i];
    }

    return weights;
}
```

statisticalOperations.h

```
#ifndef StatisticalOperations_h
#define StatisticalOperations_h

using namespace std;

class StatisticalOperations
{
    public:
      static double mean(vector<double> input);
      static double meanArray(double input[]);
      static double standardDeviation(vector<double> input , double
mean);
      static vector< vector<double> > getCovariance(vector<
vector<double> > returnVector, int size, int timeLength);
      static vector<double> getWeights(vector< vector<double> > Q,
double numberOfCompany, double noOfTargetReturn);
};

#endif
```