Imperial College Business School

MSc Financial Engineering and Risk Management – Computational Finance with C++

Markowitz Portfolio Optimization and Backtesting with C++

CID: 06014448

Author: Bach Le Hoang Nguyen

Course: Computational Finance with C++

Module Leader: Professor. Panos Parpas

Submission Date: 27 May 2025

Table of Contents

1. Introduction

Portfolio optimization plays a huge role modern investment management. The Markowitz mean-variance framework which was developed in the 1950 is remains one of the foundational models in this domain. By balancing expected return against risk, the model provides a structured way to select portfolios that lie within the efficient frontier, indicate the best possible returns for a given level of risk. The aim of this project is to implement and test the Markowitz portfolio optimization model using C++. This project's goal is to use C++ to put the Markowitz portfolio optimization model into action and evaluate it. The goal isn't simply to solve the optimization problem but it's also to provide a strong back-testing framework that shows how well such a technique would operate in a real-world. The dataset consists of daily returns for 83 FTSE 100 companies over 700 trading days. Using a rolling in-sample window of 100 days followed by a 12-day out-of-sample period, portfolios are rebalanced repeatedly to assess how performance evolves over time. A key feature of this project is the use of the Conjugate Gradient method to solve the linear system arising from the Karush-Kuhn-Tucker (KKT) conditions. The objective is to assess the model's performance across various return targets, to comprehend the risk-reward trade-off, and to illustrate performance indicators including the Sharpe ratio, realized volatility, and out-of-sample returns. This project connects theoretical finance with real algorithmic implementation, offering quantitative insights and hands-on coding expertise with C++.

2. Software Structure

This project was developed using modular C++ code to implement the Markowitz portfolio optimization model in a rolling-window backtesting framework.

- There are four main components in the program: (1) a module for reading the data file, (2) a set of functions to calculate statistics like average returns and covariances, (3) a solver that finds the optimal portfolio weights, and (4) a main program that ties everything together, runs the back-testing, and exports the results.

2.1. Logical Components and Function Structure

Component	Role	Related Functions
CSV Reader	Reads the dataset from file	readData(), string_to_double()
Data Processor	Computes mean and covariance of asset returns	computeMean(), computeCovariance()
Solver	Solves the KKT system to get portfolio weights	conjugateGradient()
Controller	Runs the full workflow step by step	main()

These functions are declared in **read_data.h** and implemented in read_data.cpp. The CSV reader class is also defined in **csv.h** and csv.cpp, and is used to parse **asset_returns.csv**

2.2. Data Input and Storage

The dataset contains the daily returns for 83 FTSE 100 companies over a period of 700 days. The function **readData()** loads this information into a 2D matrix.

2.3. Estimating Mean and Covariance

Then, the program computes two important statistics for each rolling window of 100 days, these are calculated using the given formulas:

- **computeMean()** — loops through the return matrix and averages the values
- **computeCovariance()** — measures how the returns of assets vary together

Both functions use pass-by-reference, meaning they directly update the output variables (vectors or matrices) without returning them.

2.4. Portfolio Optimization Using Conjugate Gradient

After computing the mean and covariance, the program sets up a system of equations that represent the Markowitz optimization problem with return and budget constraints.

The solver function **conjugateGradient()** takes the matrix and right-hand side, and solves for the vector of weights w. It uses an iterative algorithm that avoids matrix inversion and instead updates the solution step by step until the result is accurate enough.

2.5. Code: Step by step

The entire workflow of the program is run by the **main()** function which is located in **read_data.cpp**.

Step 1: Setup

- The program starts with setting up the environment. It defines by the inputs of the number of assets (83), the total number of return days (700), the in-sample window size (100 days), and the step size for rebalancing (12 days). Memory is then allocated for a matrix to store all return values, and the **readData()** function is called to load data from the input CSV file.

Step 2: Output File

- Next, the program prepares an output file named results.csv to log the results of each backtest step. It writes the column headers, which include the rolling window index, the target return, the out-of-sample return, portfolio risk, and the solver residual.

Step 3: Start Rolling Backtest

- The core of the program is a rolling-window backtest. It progresses through the dataset in 12-day steps, simulating a realistic rebalancing strategy. For each 100-day in-sample window, it first computes the mean return vector using **computeMean()** and then calculates the covariance matrix using **computeCovariance().**
- For every rolling window, the program then evaluates 21 different return targets, ranging from 0.0 to 0.10. For each target return:

  a) It builds the optimization system matrix and right-hand side vector.
  b) It solves the system using the **conjugateGradient()** function to obtain the optimal portfolio weights.

c) When the target return is 0.03, the system (matrix Q, vector b, and resulting weights) is exported to CSV for external verification through Octave (since MATLAB server is down)

d) It also computes the in-sample expected return and logs the residual error from the optimization.

- Following this, the program evaluates the out-of-sample performance by using the next 12 days of return data. Its re-computes the mean and covariance over that short window and applies the previously computed portfolio weights to determine the realized return and risk. These results are then saved in **results.csv**. This process repeats until the dataset is fully processed. After all windows have been evaluated, the program releases the dynamically allocated memory and exits.

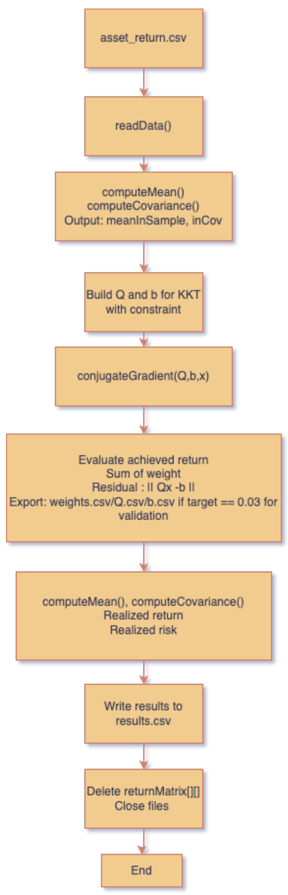


Figure 1: Diagram of the full program workflow, showing data input, parameter estimation

3. Evaluation of Portfolio Performance

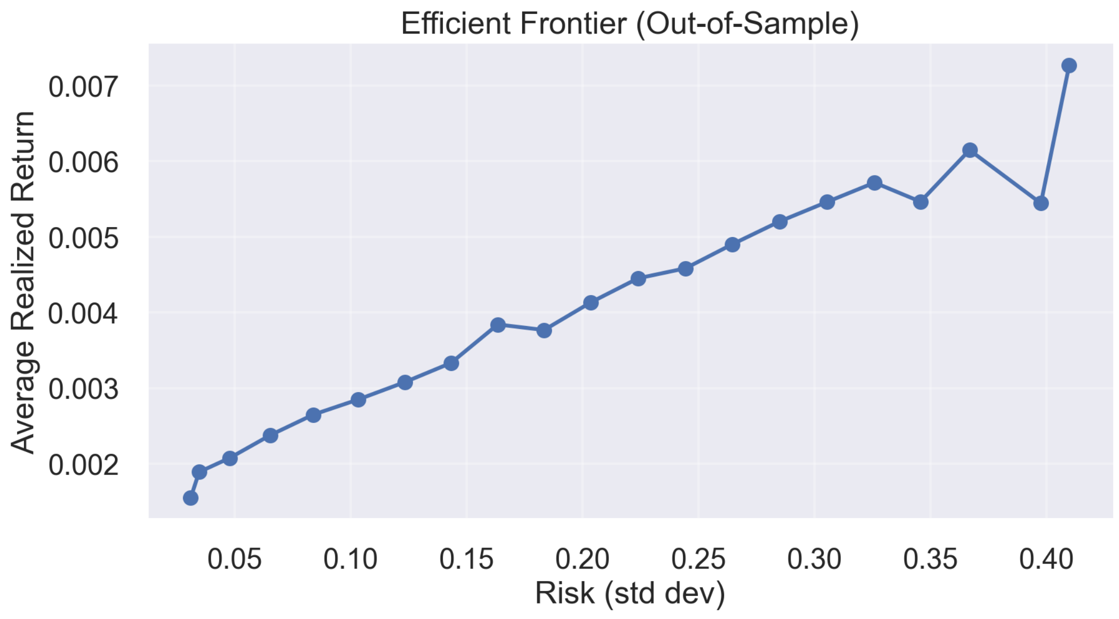
3.1. Overview

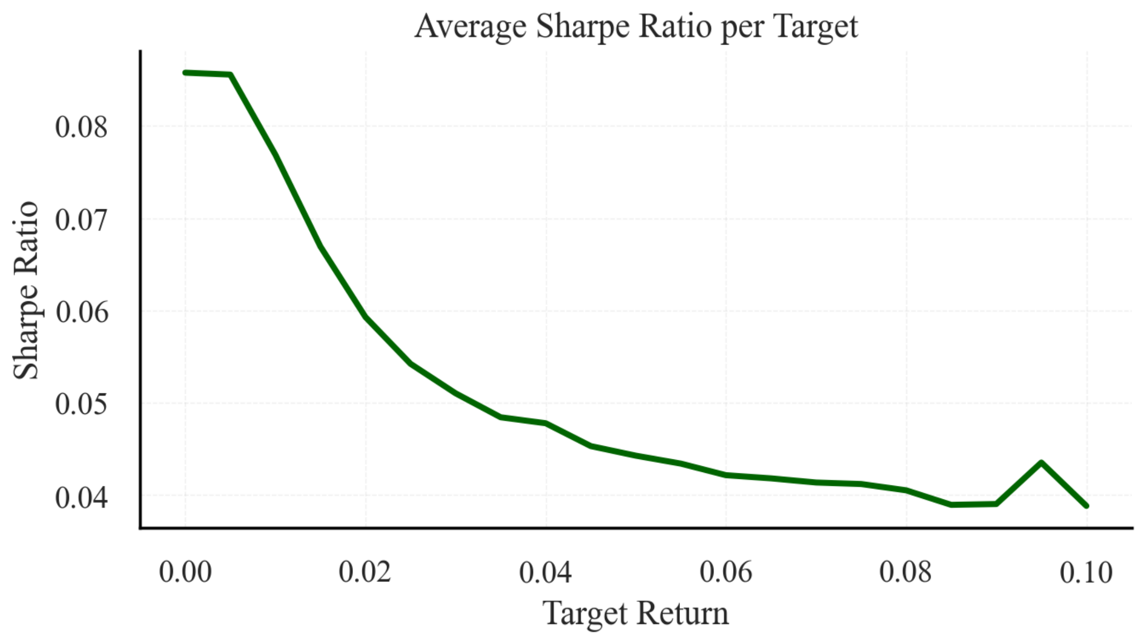
The project aims to evaluate how well the Markowitz portfolio optimization performs. So, to be able to do that, I used rolling backtest with daily returns from 83 FTSE 100 companies over 700 trading days. For each test, I built portfolios targeting 21 different return levels, from 0.0% up to 10.0%. These portfolios were updated every 12 days, based on the most recent 100 days of return data.

Three main indicators are being used to assess the strategy's efficiency and stability: out-of-sample return, standard deviation, and Sharpe ratio. These helped determine how consistent and useful the model is in a real-world scenario.

3.2. Performance Evaluation

3.2.1. Return vs. Risk (Efficient Frontier)



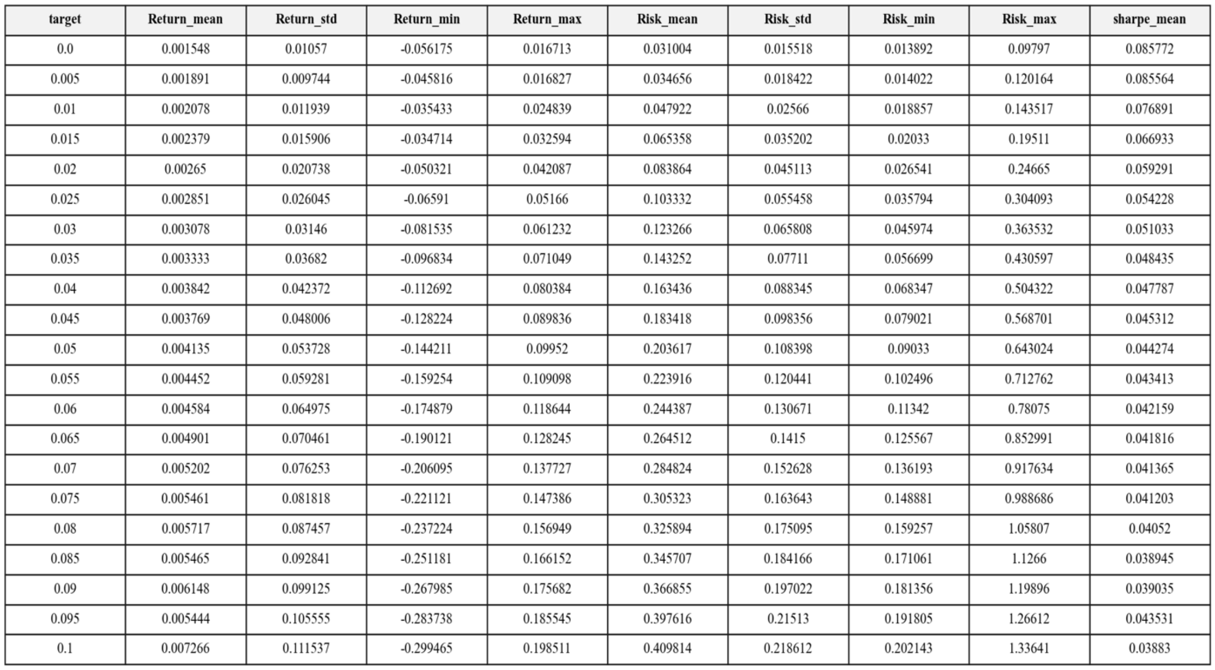


The two graphs together provide a clear view of how the Markowitz portfolio optimization performs out-of-sample. The first chart, the efficient frontier, shows the relationship between portfolio risk and realized return across a range of target return levels from 0% to 10%. As can be seen, the portfolios targeting higher returns tend to experience higher risk, creating the classic rising upward concave curve which is usually seen by modern portfolio theory.

The second chart which is the average Sharpe ratio for each target. As the model increases the target return, the Sharpe ratio begins to decline steadily, meaning risk is rising faster than return. This trend shows that there is inefficiency as we set for higher returns. Despite achieving a higher return, the risk required to reach those returns outweighs the benefit. Hence, leading to poor risk-adjusted performance. Together, these graphs suggest that the most reliable and stable portfolios lie in the mid-range of the target return, where both returns, and risk-adjusted performance are strong.

3.2.2. Summary Statistic

Table 1: Summary Statistics of Out-of-Sample Portfolio Performance

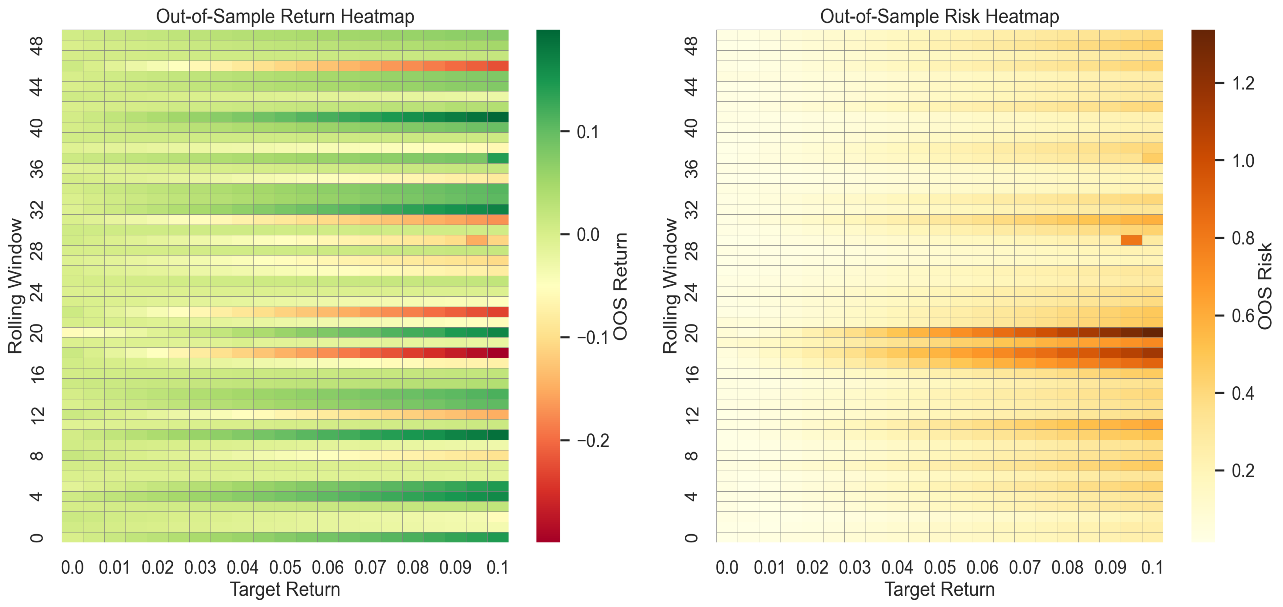


target	Return_mean	Return_std	Return_min	Return_max	Risk_mean	Risk_std	Risk_min	Risk_max	sharpe_mean
0.0	0.001548	0.01057	-0.056175	0.016713	0.031004	0.015518	0.013892	0.09797	0.085772
0.005	0.001891	0.009744	-0.045816	0.016827	0.034656	0.018422	0.014022	0.120164	0.085564
0.01	0.002078	0.011939	-0.035433	0.024839	0.047922	0.02566	0.018857	0.143517	0.076891
0.015	0.002379	0.015906	-0.034714	0.032594	0.065358	0.035202	0.02033	0.19511	0.066933
0.02	0.00265	0.020738	-0.050321	0.042087	0.083864	0.045113	0.026541	0.24665	0.059291
0.025	0.002851	0.026045	-0.06591	0.05166	0.103332	0.055458	0.035794	0.304093	0.054228
0.03	0.003078	0.03146	-0.081535	0.061232	0.123266	0.065808	0.045974	0.363532	0.051033
0.035	0.003333	0.03682	-0.096834	0.071049	0.143252	0.07711	0.056699	0.430597	0.048435
0.04	0.003842	0.042372	-0.112692	0.080384	0.163436	0.088345	0.068347	0.504322	0.047787
0.045	0.003769	0.048006	-0.128224	0.089836	0.183418	0.098356	0.079021	0.568701	0.045312
0.05	0.004135	0.053728	-0.144211	0.09952	0.203617	0.108398	0.09033	0.643024	0.044274
0.055	0.004452	0.059281	-0.159254	0.109098	0.223916	0.120441	0.102496	0.712762	0.043413
0.06	0.004584	0.064975	-0.174879	0.118644	0.244387	0.130671	0.11342	0.78075	0.042159
0.065	0.004901	0.070461	-0.190121	0.128245	0.264512	0.1415	0.125567	0.852991	0.041816
0.07	0.005202	0.076253	-0.206095	0.137727	0.284824	0.152628	0.136193	0.917634	0.041365
0.075	0.005461	0.081818	-0.221121	0.147386	0.305323	0.163643	0.148881	0.988686	0.041203
0.08	0.005717	0.087457	-0.237224	0.156949	0.325894	0.175095	0.159257	1.05807	0.04052
0.085	0.005465	0.092841	-0.251181	0.166152	0.345707	0.184166	0.171061	1.1266	0.038945
0.09	0.006148	0.099125	-0.267985	0.175682	0.366855	0.197022	0.181356	1.19896	0.039035
0.095	0.005444	0.105555	-0.283738	0.185545	0.397616	0.21513	0.191805	1.26612	0.043531
0.1	0.007266	0.111537	-0.299465	0.198511	0.409814	0.218612	0.202143	1.33641	0.03883

The summary statistics support what the graphs show visually. At lower target returns, portfolios are more stable and efficient, delivering better risk-adjusted performance. As target returns increase, returns do rise, but risk grows faster, causing a steady drop in efficiency.

The notable inefficiencies emerge in the higher target bands, particularly from 7% to 10%. Although the average return increases at these levels, the portfolio volatility rises even faster. As a result, the Sharpe ratio continues to decline, falling to its lowest point at the 10% target. This confirms the theory that was mentioned above, the minimum realized return worsens substantially, highlighting the increased downside exposure in higher-return target portfolios.

3.2.3. Portfolio Weight Characteristics



As can be observed, there are positive performance is mostly concentrated in the mid to high target return. However, there are also negative returns become more frequent and intense particularly around windows 16 to 24, where deep red zones reflect sharp underperformance. The right panel displays out-of-sample risk. A clear pattern emerges: the periods with the worst returns also exhibit the highest risk. Specifically, the same window ranges that show high negative return before. The portfolios targeting returns above 6% show volatility exceeding 1.0, indicating extreme instability. This strong alignment between high risk and poor return highlights a key breakdown in portfolio efficiency.

This instability is due to the model doesn't have constraint for short selling. Without this constraint, the optimizer often assigns large negative weights to in attempt to reach high return targets, resulting in highly leveraged and unrealistic in real world. While this can make in-sample results look attractive, but it creates significant instability when tested out-of-sample, especially during volatile periods. Overall, the heatmaps make clear that higher return targets come with even higher risk. This reinforces again, the conclusion that moderate return targets offer a better balance of performance and stability, while aggressive strategies lead to unreliable and risk-heavy outcomes.

3.2.4. Solver Performance: Conjugate Gradient Method

ans (final residual norm from Octave CG)	1.1083e-07
||Q * x_cpp - b||	6.45414e-06
||x_cpp - x_direct||	0.164557
||x_cpp - x_octave||	0.0277488

To verify the accuracy of the Conjugate Gradient (CG) solver implemented in C++, a validation test was conducted using Octave. The solution vector generated (sol) contains 83 asset weights and includes both positive and negative values, which is consistent with the model's unconstrained short-selling framework. The final residual norm reported by Octave was 1.1083×10^{-7}, indicating that the CG method successfully reached a stable and precise solution. On the other side, the C++ implementation produced a similarly low residual of $\left|Qx_{\text{cpp}} - b\right| = 6.4541 \times 10^{-6}$, confirming that the solution accurately satisfies the linear system and satisfy the requirement by the coursework.

Beyond just checking residuals, a direct comparison was made between the solution from the C++ CG solver and two other benchmarks: Octave's exact solution obtained via matrix inversion, and Octave's own CG solver. The difference between the C++ CG solution and the exact direct solution was approximately 0.1646, while the difference between the C++ and Octave CG solutions was much smaller at 0.0277. This means the C++ implementation behaves consistently with standard numerical methods, with only minor numerical deviations which could be due to iterative precision or floating-point operations. Overall, the test confirms that the C++ solver is both accurate and stable.

4. Conclusion

The model performs logically in most cases. However, there are a few problems during testing and analysis. The first issue relates to how the optimizer handles aggressive return targets. When we aim for high returns like above 5%, the model would start allocating extreme weights, including large negative positions. This behavior can be seen in the C++ terminal output, where it would show weights like –1.5, –2.5, and in some cases, even more extreme. These shows the existence of short positions under the unconstrained Markowitz framework, but they are unrealistic in practice. In addition, other issue is that the model relies on just 100 days of past data to decide which assets to invest in. It's because of the uneven amount of assets and data, the results can be misleading. This becomes a bigger problem when aiming for higher returns, since the model puts more weight on assets that happened to perform well recently. While the portfolio often meets the return target in the sample period, it usually performs worse afterward, as some of the OOS_r outputs in the terminal are very negative. Hence, the actual return even drops below zero, showing that the model is focused on short-term patterns that make it become inefficient. This leads to less reliable performance and greater risk, especially in out-of-sample testing.

Overall, the model is stable and accurate. The residuals calculated by the Conjugate Gradient are consistently within the tolerance of 1e-6, which confirms that the numerical solution is valid, and that the optimization is solving the system correctly.

There are a few simple ways to make the model more realistic. First, implementing the limit of how much short selling it can do or completely constrain short selling, so the portfolio would avoid extreme position. Second, it could use better methods for measuring risk like shrinkage estimator, Minimum-Variance Matrix Adjustment to smooth out noisy data. Finally, adding some penalty for frequent trading and including transaction costs would help the model reflect real-world conditions and avoid rebalancing between aggressive positions.

5. Appendix: Source Code

5.1. read_data.h – Function declarations

```
#ifndef READ_DATA_H
#define READ_DATA_H

#include <string>
#include <vector>

double string_to_double(const std::string& s);
void readData(double** data, const std::string& fileName);

void computeMean(double** data,int numberAssets,int t_start,int
window,std::vector<double>& mean);

void computeCovariance(double** data,int numberAssets,int t_start,int window,const
std::vector<double>& mean,std::vector<std::vector<double> >& cov);

void conjugateGradient(const std::vector<std::vector<double> >& Q,const
std::vector<double>& b,std::vector<double>& x,double epsilon = 1e-6);

#endif
```

5.2. read_data.cpp – Main program logic + CG solver

```
#include "read_data.h"
#include "csv.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <cmath>
#include <numeric>


using namespace std;


//g++ -c read_data.cpp
// g++ -c csv.cpp
// g++ -o portfolioSolver csv.o read_data.o
// ./portfolioSolver


double string_to_double(const std::string& s) {
    istringstream i(s);
    double x;
    if (!(i >> x))
        return 0.0;
    return x;
}

void readData(double **data, const std::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);}
                                                                }
void computeMean(double** data,int numberAssets,int t_start,int
window,vector<double>& mean) {
    for (int i = 0; i < numberAssets; ++i) {
        double sum = 0.0;
        for (int j = 0; j < window; ++j)
            sum += data[i][t_start + j];
        mean[i] = sum / window;
    }
}

void computeCovariance(double** data,int numberAssets,int t_start,int window,const
vector<double>& mean,vector<vector<double> >& cov) {
    for (int i = 0; i < numberAssets; ++i) {
        for (int j = 0; j < numberAssets; ++j) {
            double sum = 0.0;
            for (int k = 0; k < window; ++k) {
                double ri = data[i][t_start + k] - mean[i];
                double rj = data[j][t_start + k] - mean[j];
                sum += ri * rj;
            }
            cov[i][j] = sum / (window - 1);
        }
    }
}

void conjugateGradient(const vector< vector<double> >& Q,const vector<double>&
b,vector<double>& x,double epsilon) {
    int n = Q.size();

    vector<double> r(n);
    vector<double> p(n);
    vector<double> Qp(n);

    // r0 = b - Q * x0
    for (int i = 0; i < n; ++i) {
        double Qxi = 0.0;
        for (int j = 0; j < n; ++j){
            Qxi += Q[i][j] * x[j];
        }
        r[i] = b[i] - Qxi;
        p[i] = r[i];
    }

    double rs_old = 0.0;
    for (int i = 0; i < n; ++i){
        rs_old += r[i] * r[i];

    }

    for (int k = 0; k < n; ++k) {

        // Qp_k = Q * p_k
        for (int i = 0; i < n; ++i) {
            Qp[i] = 0.0;
            for (int j = 0; j < n; ++j)
                Qp[i] += Q[i][j] * p[j];
        }

        double alpha_num = rs_old;
        double alpha_den = 0;
        for (int i = 0; i < n; ++i){
            alpha_den += p[i] * Qp[i];
        }
        double alpha = alpha_num / alpha_den;
        //  x = x + alpha * p
        for (int i = 0; i < n; ++i)
            x[i] = x[i] + alpha * p[i];

        // r = r - alpha * Qp
        for (int i = 0; i < n; ++i){
            r[i] -= alpha * Qp[i];
        }


        double rs_new = 0.0;
        for (int i = 0; i < n; ++i){
            rs_new += r[i] * r[i];
        }

        if (sqrt(rs_new) < epsilon){
            break;
        }
        // beta_k = (r_(k+1)^T * r_(k+1)) / (r_k^T * r_k)
        double beta = rs_new / rs_old;

        // p_(k+1) = r_(k+1) + beta_k * p_k
        for (int i = 0; i < n; ++i)
            p[i] = r[i] + beta * p[i];

        rs_old = rs_new;
    }
}

int main() {
    const int numberAssets = 83;
    const int numberReturns = 700;
    const int window = 100;
    const int step = 12;
    const int numberTargets = 20;

    double** returnMatrix = new double*[numberAssets]; // a matrix to store the
return data
    //allocate memory for return data
    for (int i = 0; i < numberAssets; ++i)
        returnMatrix[i] = new double[numberReturns];

    //read the data from the file and store it into the return matrix
    string fileName="asset_returns.csv";
    readData(returnMatrix, fileName);
    // returnMatrix[i][j] stores the asset i, return j value
    vector<double> meanInSample(numberAssets);
    vector< vector<double> > inCov(numberAssets, vector<double>(numberAssets));

    ofstream results("results.csv");
    results << "window,target,Return,Risk, Residual\n";


    // Rolling backtest
    int rollingWindow = 0;
    for (int t_start = 0; t_start + window + step <= numberReturns; t_start += step)
{
        computeMean(returnMatrix, numberAssets, t_start, window, meanInSample);
        computeCovariance(returnMatrix, numberAssets, t_start, window, meanInSample,
inCov);
        for (int t = 0; t <= numberTargets; ++t) {
            double target = t * 0.10 / numberTargets;
            int dimension = numberAssets + 2;


            vector< vector<double> > Q(dimension, vector<double>(dimension, 0.0));
            vector<double> b(dimension, 0.0);


            for (int i = 0; i < numberAssets; ++i)
                for (int j = 0; j < numberAssets; ++j)
                    Q[i][j] = inCov[i][j];

            for (int i = 0; i < numberAssets; ++i) {
                double ri = meanInSample[i];
                Q[i][numberAssets] = -ri;
                Q[i][numberAssets + 1] = -1.0;
                Q[numberAssets][i] = -ri;
                Q[numberAssets + 1][i] = -1.0;
            }
            b[numberAssets] = -target;
            b[numberAssets + 1] = -1.0;
            vector<double> weights(dimension, 1.0);
            conjugateGradient(Q, b, weights, 1e-6);




            // CSV Export for validation
            if (fabs(target - 0.03) < 1e-6) {
                ofstream outW("weights.csv");
                for (int i = 0; i < dimension; ++i)
                    outW << weights[i] << "\n";
                outW.close();
                ofstream outB("b.csv");
                for (int i = 0; i < dimension; ++i)
                    outB << b[i] << "\n";
                outB.close();

                ofstream outQ("Q.csv");
                for (int i = 0; i < dimension; ++i) {
                    for (int j = 0; j < dimension; ++j) {
                        outQ << Q[i][j];
                        if (j < dimension - 1) outQ << ",";
                    }
                    outQ << "\n";
                }
                outQ.close();
            }
            cout << "[CG] sample weights: ";
            for (int i = 0; i < 5; ++i) cout << weights[i] << "  ";
            cout << "\n";

            // Check

            double sumw = 0.0;
            for (int i = 0; i < numberAssets; ++i)
                sumw += weights[i];
            cout << "Sum of weight = " << sumw << "\n";

            double actualReturn = 0.0;
            for (int i = 0; i < numberAssets; ++i)
                actualReturn += meanInSample[i] * weights[i];
            cout << "Actual return = " << actualReturn << ", Target = " << target <<
"\n";


            vector<double> Qx(dimension, 0.0);
            for (int i = 0; i < dimension; ++i)
                for (int j = 0; j < dimension; ++j)
                    Qx[i] += Q[i][j] * weights[j];
            double residual = 0.0;
            for (int i = 0; i < dimension; ++i)
                residual += pow(Qx[i] - b[i], 2);
            residual = sqrt(residual);
            cout << "Residual 'Qx - b' = " << residual << "\n";

            // Out-of-sample
            vector<double> meanOutSample(numberAssets);
            vector< vector<double> > outCov(numberAssets,
vector<double>(numberAssets));
```

```
            computeMean(returnMatrix, numberAssets, t_start + window, step,
meanOutSample);
            computeCovariance(returnMatrix, numberAssets, t_start + window, step,
meanOutSample, outCov);

            double realizedReturn = 0.0;
            for (int i = 0; i < numberAssets; ++i)
                realizedReturn += meanOutSample[i] * weights[i];
            double realizedVar = 0.0;
            for (int i = 0; i < numberAssets; ++i)
                for (int j = 0; j < numberAssets; ++j)
                    realizedVar += weights[i] * outCov[i][j] * weights[j];
            double realizedRisk = sqrt(realizedVar);

            results << rollingWindow << "," << target << "," << realizedReturn <<
"," << realizedRisk << "," << residual << "\n";

            cout << "Window["<< t_start << "] target="<< target
                 << ", OOS_r="<< realizedReturn
                 << ", sigma="<< realizedRisk << "\n";
        }
        rollingWindow++;

    }

    results.close();

    // Cleanup
    for (int i = 0; i < numberAssets; ++i)
        delete[] returnMatrix[i];
    delete[] returnMatrix;

    return 0;
}
```

5.3. conjgrad.m – MATLAB reference implementation for CG

```
function [x] = conjgrad(A, b, x)
    r = b - A * x;
    p = r;
    rsold = r' * r;

    for i = 1:length(b)
        Ap = A * p;
        alpha = rsold / (p' * Ap);
        x = x + alpha * p;
        r = r - alpha * Ap;
        rsnew = r' * r;
        if sqrt(rsnew) < 1e-10
              break;
        end
        p = r + (rsnew / rsold) * p;
        rsold = rsnew;
    end
end
```

5.4. testSolutions.m – Validation script

```
clear;
data=load('asset_returns.csv');
assets=83;
tR=0.03;
means=zeros(assets,1);
for ii=1:assets
means(ii)=mean(data(:,ii));
%result=sprintf('%0.9f\n',means(ii));
%fprintf(result)
end

covMat=cov(data);
e=ones(assets,1);

A=[covMat,-means,-e; -means',0,0;-e',0,0];
rh=[zeros(assets,1);-tR;-1];

sol=A\rh

x=ones(assets+2,1);
x=conjgrad(A, rh, x);

norm(sol-x)


%eigs(covMat,20,'sm')

%length(covMat)

Q     = csvread('Q.csv');
b     = csvread('b.csv');
x_cpp = csvread('weights.csv');
```

```
x_direct = Q \ b;
x_oct    = conjgrad(Q, b, ones(size(b)));

fprintf("||Q*x_cpp - b||       = %g\n", norm(Q*x_cpp - b));
fprintf("||x_cpp - x_direct|| = %g\n", norm(x_cpp - x_direct));
fprintf("||x_cpp - x_octave|| = %g\n", norm(x_cpp - x_oct));
```