Imperial College
London

COMPUTATIONAL FINANCE WITH C++

IMPERIAL COLLEGE BUSINESS SCHOOL

DEPARTMENT OF FINANCE

Markowitz Model & Rolling Window Back-Testing

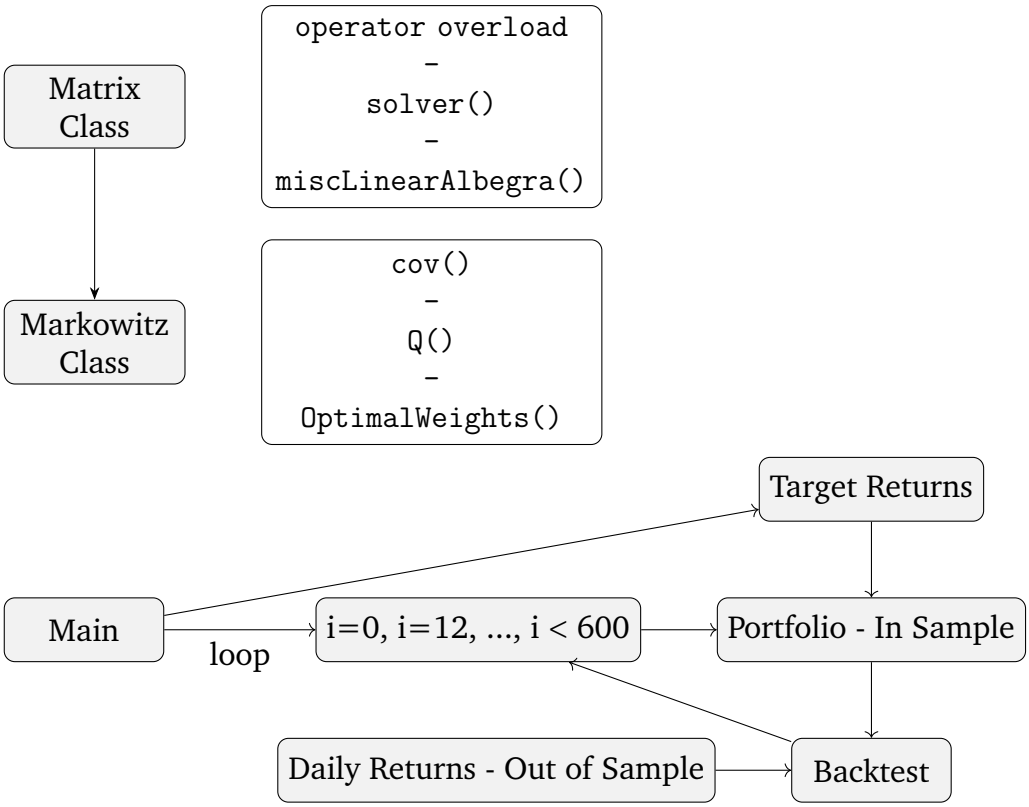
Author:
Edward Peterson (CID: 01502703)

Date: June 1, 2024

1 Software Structure

www.github.com/ep4518/CFcrsw

- no use of polymorphism
- read_data.h and read_data.cpp unchanged
- defined type Vector and Lattice as vector<double> and vector<vector<double>>
- defined class "Matrix" holds a lattice and implements rudimentary linear algebra with multiple constructors available e.g. (rows, columns), (Lattice), ().
  - operator overload for multiplication, addition, subtraction, unary negative and also for scalar equivalent operations
  - operator overload for splcing, along with functionallity for insertion, printing, retrieval, shape etc.
  - ultimately building towards implementing the Conjugate Gradient Descent Solver.
- implemented numpy-like horizontal and vertical stacking of Matrix triples
- Markowitz class for defining a portfolio with optimal asset weightings
  - mean() - average returns for each asset over sample period - $\bar{r}_i = \frac{1}{n}\Sigma_{k=1}^{n} r_{i,k}$
  - cov() - covariance of asset returns - $\Sigma_{ij} = \frac{1}{n-1}\sum_{k=1}^{n}(r_{i,k} - \bar{r}_i)(r_{j,k} - \bar{r}_j)$
  - b(double target_return), Q() - vstack(hstack, hstack, hstack)
  - optimal_weights(): $Qx = b$
- backtesting function implemented for each iteration in the rolling window
- wrote array of result structs in csv form for plotting with python - write_data.h



2 Evaluation

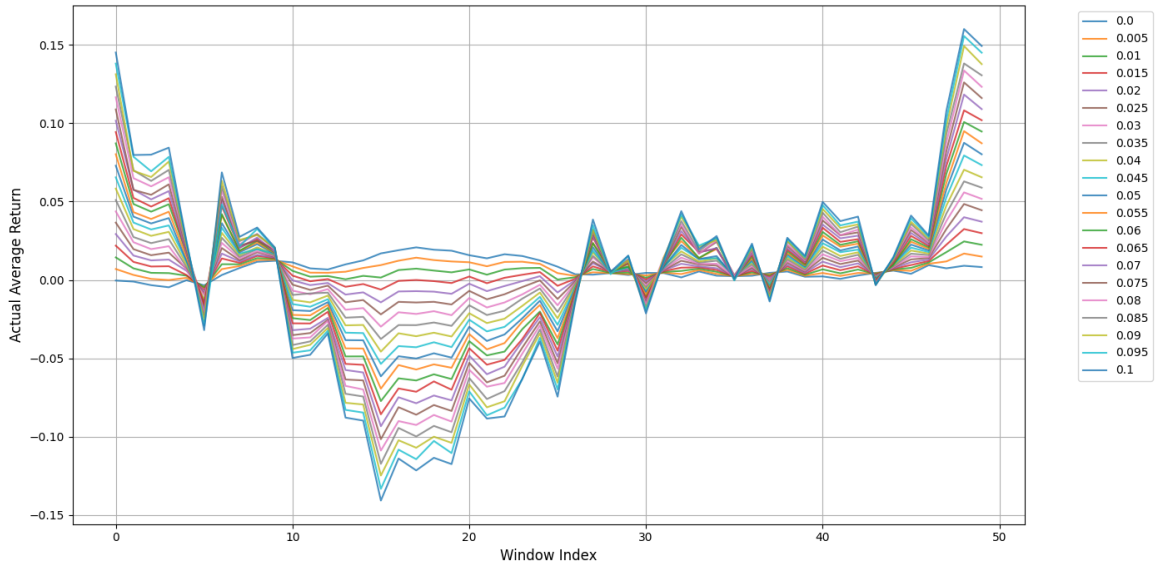


Figure 1: Realised Rolling Window Average OOS Return for each Target

Fig. 1 portrays the rolling window realised average return for each target over the course of the backtest. There are 50 periods in the backtest indexed on the x-axis of the plot. Out of sample performance of the Markowtiz model is poor. The cumulative performance over the enitre domain is slightly better the larger the target return becomes (Fig. 2).

To evaluate the accuracy of our Markowitz implementation (Fig. 1), we contrast with a numpy analog, where the linalg library provides a relaible solver to compare the accuracy of the conjugate gradient descent algorithm (CGD) in cpp. Fig. 6 shows the drastic impact of the fractional differences in optimal weights we see whilst solving with CGD, on the overall performance during the backtesting phase. Following extensive debugging, these differences were also observed in the Matlab conjgrad.m implementation. Any CGD solver used had $||Qx - b|| \approx e^{-5}$ for the range of target returns during backtesting (Fig. 7), where as for prebuilt solvers the norm value usually lies in the range $\approx e^{-11}$.

In both cases, the Markowitz portfolio effectively captures the market Beta, and produces returns that scale consistently with the amount of risk taken by the investor (Fig. 3) (where the magnitude and direction of those returns is dictated by those of the market). Fig. 4 demonstrates the Sharpe ratio of the portfolios for an example backtest period. In general, we observed this classic efficient frontier shape in most periods, allowing for inversion in periods of negative market returns.

Fig. 5 portrays a cumulative portfolio return next to its constituent assets. As with Fig. 4, this figure is a product of selection bias.

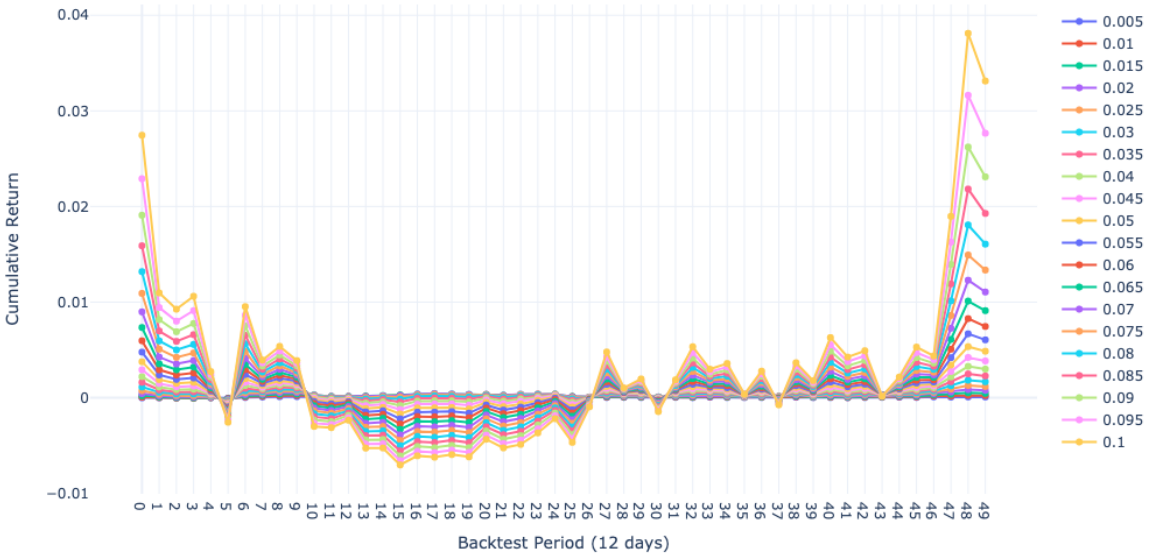


Figure 2: Cumulative 12-day Out of Sample Returns of Optimal Portfolios (No Frictions)

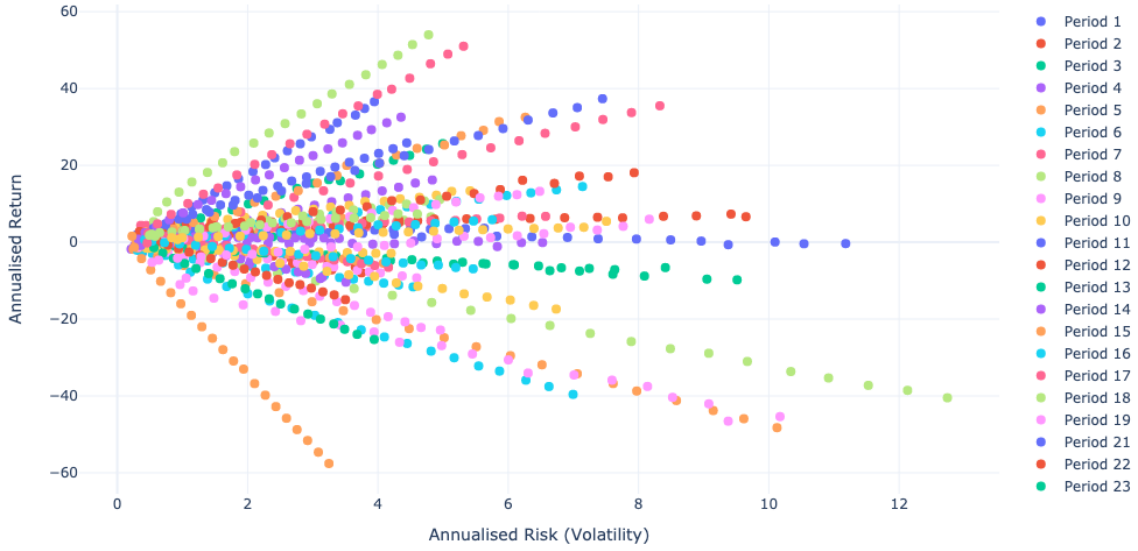


Figure 3: Efficient Frontier with Optimal Portfolios (All Backtest Periods [1,50])

3 Appendix

Figure 4: Efficient Frontier with Optimal Portfolios (Backtest Period 9)

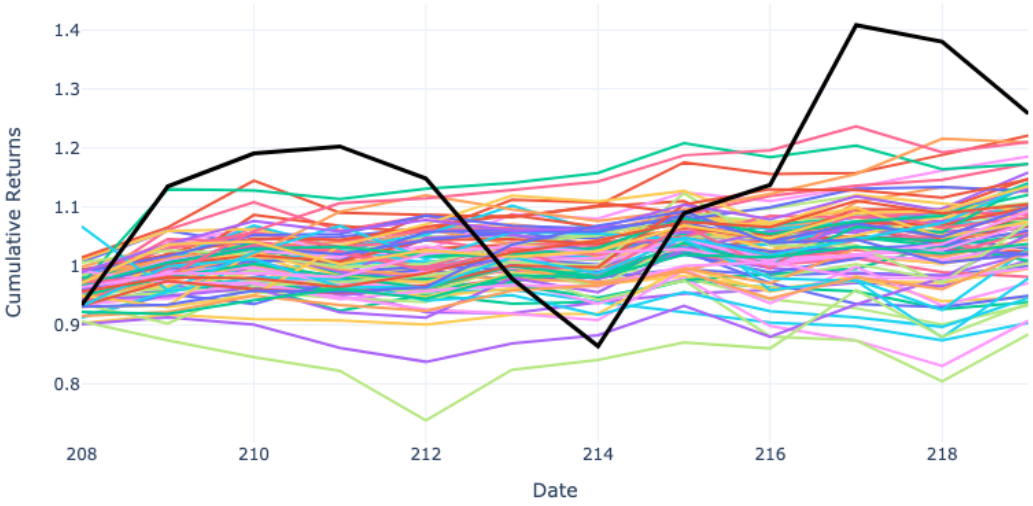


Figure 5: Cumulative Returns of Optimized Portfolio vs. Assets (Backtest Period 9, Target R 0.04)

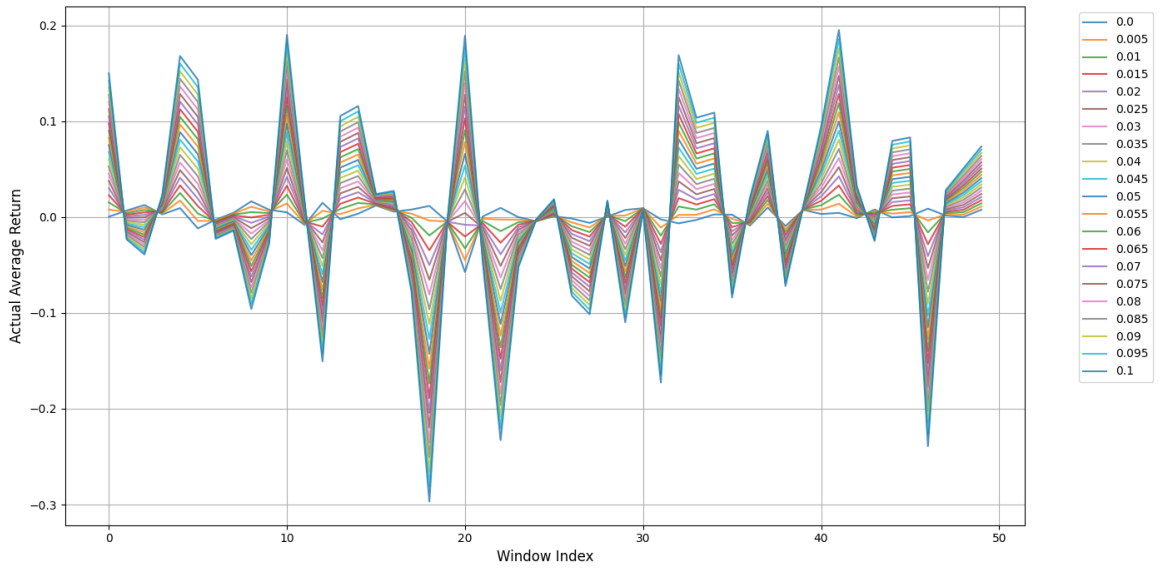


Figure 6: Realised Rolling Window Average OOS Return for each Target - Python

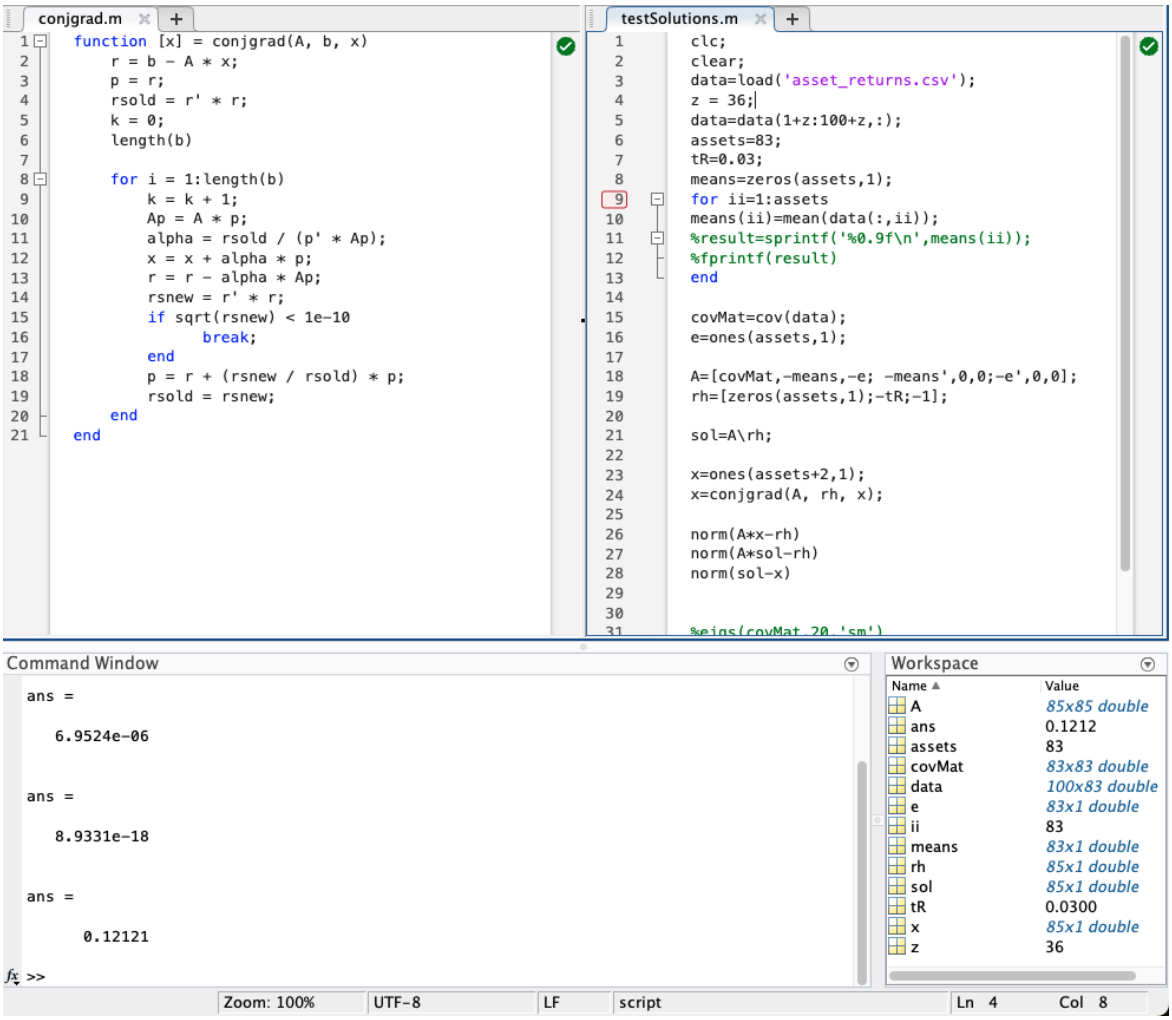


Figure 7: Matlab

3.1 Code

www.github.com/ep4518/CFcrsw

- main.cpp
- csv.cpp
- read_data.cpp
- linalg.cpp
- Markowitz.cpp
- write_data.cpp

3.1.1 main.cpp

```
// main.cpp
#include "read_data.h"
#include "write_data.h"
#include "linalg.h"
#include "Markowitz.h"
#include <iostream>

using namespace std;

// A function to implement a backtest for each window. Takes as input the optimal weights from the IS returns,
// the Out of Sample returns, and the target_returns which is linspace(0,21,0.1)
Matrix back_testing(const Matrix &optimal_weights, const Matrix &OOS_returns, const Matrix &target_returns);

// command line arguments unused. Please use cmake to compile and run.
int  main (int  argc, char  *argv[])
{

    int numberAssets=83;
    int numberReturns=700;

    // Lattice is defined in linalg.h, as is Vector
    Lattice returnMatrix(numberAssets, Vector(numberReturns)); // a matrix to store the return data

    //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

    Matrix daily_returns(returnMatrix); // (Assets -> r) * (Days -> c) == 83 * 700

    // np.linspace(0,21,0.1) ==
    Lattice tr =  {
            {0.   , 0.005, 0.01 , 0.015, 0.02 , 0.025, 0.03 , 0.035, 0.04 ,
            0.045, 0.05 , 0.055, 0.06 , 0.065, 0.07 , 0.075, 0.08 , 0.085,
            0.09 , 0.095, 0.1}
    };

    // Matrix object constructed with Lattice => see linalg.h
    Matrix target_returns(tr);

    Matrix daily_returns_test = daily_returns(0, numberAssets, 360, 460);
    Markowitz portfolio(daily_returns_test, target_returns);
    portfolio.NormTest();

    // results array of size number of windows of Result structs.
    // Result struct defined in write_data.h
    Result results[50];
```

```
    for (int i = 0; i < numberReturns - 100; i += 12) {
    // iterating over the starts of each in sample window
        int index = int(i/12);
        // for the results array
        cout << "Moving to index " << index << endl;
        int start = index, mid = index + 100, end = index + 112; // start of IS, start of OOS, end of OOS
        Matrix daily_returns_IS = daily_returns(0, numberAssets, start, mid); // splice => see linalg.h
        Matrix daily_returns_OOS = daily_returns(0, numberAssets, mid, end);
        Markowitz portfolio(daily_returns_IS, target_returns);     // Markowitz class => see Markowitz.h
        Matrix df_optimal_weights = portfolio.getWeights();
        // calculate optimal porfolio weights using CGD method of Matrix object
        // df_act_returns is 21 * 3 Matrix object that contains target return,
        // actual return and portfolio covariance for each target return
        Matrix df_act_returns = back_testing(df_optimal_weights, daily_returns_OOS, target_returns);
        results[index] = {
                index,
                df_act_returns,
                df_optimal_weights
        };
    }

    results[0].back_test.prn();

    write_data(results, 50);

    return 0;
}

Matrix back_testing(const Matrix &optimal_weights, const Matrix &OOS_returns, const Matrix &target_returns) {
    int num_targ_rets = optimal_weights.getRows();
    int num_assets = OOS_returns.getRows();
    Markowitz OOS_rets(OOS_returns, target_returns);
    // Result matrix to store performance for each set of weights
    Matrix results(num_targ_rets, 3);

    for (int i = 0; i < num_targ_rets; i++) {
        // Extracts the i-th row of optimal weights
        Matrix optimal_weights_row(num_assets, 1);
        for (int j = 0; j < num_assets; j++) {
            optimal_weights_row.insert(j, 0, optimal_weights(i, j));
        }
        Matrix act_ave_return = OOS_rets.mean().transpose() * optimal_weights_row;
        Matrix pf_cov = optimal_weights_row.transpose() * OOS_rets.cov() * optimal_weights_row;
        results.insert(i, 0, target_returns(0,i));
        results.insert(i, 1, act_ave_return(0,0));
        results.insert(i, 2, pf_cov(0,0));
    }
    return results;
}
```

3.1.2 csv.cpp

```
//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];
}
```

3.1.3 read_data.cpp

```
//read_data.h
#ifndef _READ_DATA_H
#define _READ_DATA_H
#include <sstream>
#include <vector>

double string_to_double( const std::string& s );
void readData(std::vector<std::vector<double> >& returnMatrix,std::string fileName);

#endif

//read_data.cpp
#include <fstream>
#include <stdlib.h>
#include <sstream>
#include <vector>
#include "csv.h"
```

```
//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 )
{
 std::istringstream i(s);
 double x;
 if (!(i >> x))
  return 0;
 return x;
}

void readData(std::vector<std::vector<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);}
}
```

3.1.4 linalg.cpp

```
// linalg.h
#ifndef _LINALG_H
#define _LINALG_H

#include <vector>
#include <iostream>

typedef std::vector<double> Vector;
typedef std::vector<Vector> Lattice;

class Matrix {
private:
    int rows, columns;
    Lattice M;
public:
    // Default constructor
    Matrix() : rows(0), columns(0), M() {}

    // Construct from pre-existing Lattice
    Matrix(Lattice _M) {M = _M; rows = _M.size(); columns = _M[0].size();}

    // Construct a zero matrix with given dimensions
    Matrix(int _rows, int _columns) : rows(_rows), columns(_columns), M(_rows, Vector(_columns, 0.0)) {}

    // Access element (const version)
    const double& operator()(size_t i, size_t j) const { return M[i][j]; }

    // Method to insert an element into the matrix
    void insert(size_t i, size_t j, double value) {
```

```
        if (i >= rows || j >= columns) {
            throw std::out_of_range("Matrix indices out of range");
        }
        M[i][j] = value;
    }

    // pandas like shape method
    void shape() {printf("(%d, %d)\n",rows, columns);}

    // Get number of rows
    int getRows() const { return rows; }

    // Get number of columns
    int getColumns() const { return columns; }

    // Get the underlying matrix
    const Lattice& getMatrix() const { return M; }

    // Access element (non-const version)
    double& operator()(size_t i, size_t j) {
        if (i >= rows || j >= columns) throw std::out_of_range("Matrix indices out of range");
        return M[i][j];
    }

    // Unary minus operator for matrix
    Matrix operator-() const;

    // Splicing operator
    Matrix operator()(int row_start, int row_end, int col_start, int col_end) const;

    Matrix transpose() const;

    // Skip inversion. Utilise conjugate gradient algorithm (see .pdf/ wiki)
    // Mx = b => x*
    //  https://en.wikipedia.org/wiki/Conjugate_gradient_method
    Matrix solver(const Matrix &b, const double tol, const int debug) const;

    // Matrix multiplication
    friend Matrix operator*(const Matrix& A, const Matrix& B);
    // Scalar multiplication
    friend Matrix operator*(const double& a, const Matrix&A);

    // Matrix addition/ subtraction
    friend Matrix operator+(const Matrix& A, const Matrix& B);
    friend Matrix operator-(const Matrix& A, const Matrix& B);

    // Dot product of two column matrices
    double dot(const Matrix& A) const;

    // Print the matrix
    void prn() const;

    double norm();

};

// hstack and vstack for constructing Q
Matrix vstack(const Matrix &A, const Matrix &B, const Matrix &C);
Matrix hstack(const Matrix &A, const Matrix &B, const Matrix &C);

#endif

//linalg.cpp
#include "linalg.h"
#include <cmath>
#include <stdexcept>
#include <iostream>

void Matrix::prn() const {
```

```
    std::cout << "[\n";
    for (const Vector& row : M) {
        std::cout << " [ ";
        for (const double& elem : row) {
            std::cout << elem << " ";
        }
        std::cout << "]\n";
    }
    std::cout << "]\n";
}

Matrix Matrix::operator()(int row_start, int row_end, int col_start, int col_end) const {
    if (row_start < 0 || row_end > rows || col_start < 0 || col_end > columns) {
        throw std::out_of_range("Index out of bounds");
    }
    Lattice result;
    for (int i = row_start; i < row_end; ++i) {
        Vector row;
        for (int j = col_start; j < col_end; ++j) {
            row.push_back(M[i][j]);
        }
        result.push_back(row);
    }
    return Matrix(result);
}

// Transpose of the matrix
Matrix Matrix::transpose() const {
    Lattice transposed(columns, Vector(rows));
    for (size_t i = 0; i < rows; ++i) {
        for (size_t j = 0; j < columns; ++j) {
            transposed[j][i] = M[i][j];
        }
    }
    return {transposed};
}

// Conjugate Gradient Solver
Matrix Matrix::solver(const Matrix &b, const double tol, const int debug) const {
    if (M.empty() || rows != columns) {
        throw std::invalid_argument("Matrix dimensions are not compatible for inversion");
    }
    if (b.getColumns() != 1 || b.getRows() != rows) {
        throw std::invalid_argument("b must be a column vector with the same number of rows as the matrix.");
    }
    Matrix x(Lattice(rows, Vector(1, 1.0)));
    Matrix r = b - (*this) * x;
    Matrix p = r;
    double rsold = r.dot(r);
    double rsnew;
    if (debug == 1) {
        std::cout << "x.norm() " << x.norm() << std::endl;
        std::cout << "r.norm() " << r.norm() << std::endl;
        for (int k = 0; k < b.getRows(); ++k) {
            Matrix Ap = (*this) * p;
            std::cout << "Ap.norm() " << Ap.norm() << std::endl;
            double alpha = rsold / p.dot(Ap);
            std::cout << "alpha " << alpha << std::endl;
            x = x + alpha * p;
            std::cout << "x.norm() " << x.norm() << std::endl;
            r = r - alpha * Ap;
            std::cout << "r.norm() " << r.norm() << std::endl;
            rsnew = r.dot(r);
            std::cout << "rsnew " << rsnew << std::endl;
            if (std::sqrt(rsnew) < tol) break;
            p = r + (rsnew / rsold) * p;
            std::cout << "p.norm() " << p.norm() << std::endl;
            rsold = rsnew;
            std::cout << "rsold " << rsold << std::endl;
```

```
        }
    }
    else {
        for (int k = 0; k < b.getRows(); ++k) {
            Matrix Ap = (*this) * p;
            double alpha = rsold / p.dot(Ap);
            x = x + alpha * p;
            r = r - alpha * Ap;
            rsnew = r.dot(r);
            if (std::sqrt(rsnew) < tol) break;
            p = r + (rsnew / rsold) * p;
            rsold = rsnew;
        }
    }

    return x;
}

// Matrix multiplication
Matrix operator*(const Matrix& A, const Matrix& B) {
    if (A.getColumns() != B.getRows()) {
        throw std::invalid_argument("Matrix dimensions must agree for multiplication.");
    }
    Matrix result(A.getRows(), B.getColumns());
    for (int i = 0; i < A.getRows(); ++i) {
        for (int j = 0; j < B.getColumns(); ++j) {
            for (int k = 0; k < A.getColumns(); ++k) {
                result(i, j) += A(i, k) * B(k, j);
            }
        }
    }
    return result;
}

// Matrix addition
Matrix operator+(const Matrix& A, const Matrix& B) {
    if (A.getRows() != B.getRows() || A.getColumns() != B.getColumns()) {
        throw std::invalid_argument("Matrix dimensions must agree for addition.");
    }
    Matrix result(A.getRows(), A.getColumns());
    for (int i = 0; i < A.getRows(); ++i) {
        for (int j = 0; j < A.getColumns(); ++j) {
            result(i, j) = A(i, j) + B(i, j);
        }
    }
    return result;
}

// Matrix subtraction
Matrix operator-(const Matrix& A, const Matrix& B) {
    if (A.getRows() != B.getRows() || A.getColumns() != B.getColumns()) {
        throw std::invalid_argument("Matrix dimensions must agree for subtraction.");
    }
    Matrix result(A.getRows(), A.getColumns());
    for (int i = 0; i < A.getRows(); ++i) {
        for (int j = 0; j < A.getColumns(); ++j) {
            result(i, j) = A(i, j) - B(i, j);
        }
    }
    return result;
}

// Unary - operator for matrix
Matrix Matrix::operator-() const {
    Matrix result(rows, columns);
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < columns; ++j) {
            result(i, j) = -M[i][j];
        }
```

```
    }
    return result;
}

// Scalar multiplication
Matrix operator*(const double& a, const Matrix& A) {
    Matrix result(A.getRows(), A.getColumns());
    for (int i = 0; i < A.getRows(); ++i) {
        for (int j = 0; j < A.getColumns(); ++j) {
            result(i, j) = a * A(i, j);
        }
    }
    return result;
}

// Dot product of two column matrices
double Matrix::dot(const Matrix& A) const {
    if (columns != 1 || A.getColumns() != 1 || rows != A.getRows()) {
        throw std::invalid_argument("Dot product requires column matrices of the same size.");
    }
    double result = 0.0;
    for (int i = 0; i < rows; ++i) {
        result += M[i][0] * A(i, 0);
    }
    return result;
}

double Matrix::norm() {
    double result = 0.0;
    for (int i = 0; i < rows; i++) {
        result += pow(M[i][0], 2.0);
    }
    return sqrt(result);
}

// Vertical stack of three matrices
Matrix vstack(const Matrix &A, const Matrix &B, const Matrix &C) {
    if (A.getColumns() != B.getColumns() || A.getColumns() != C.getColumns()) {
        throw std::invalid_argument("vstack requires matrices of the same width.");
    }
    int h = A.getRows() + B.getRows() + C.getRows();
    Matrix result(h, A.getColumns());

    for (int i = 0; i < A.getRows(); ++i) {
        for (int j = 0; j < A.getColumns(); ++j) {
            result(i, j) = A(i, j);
        }
    }
    for (int i = 0; i < B.getRows(); ++i) {
        for (int j = 0; j < B.getColumns(); ++j) {
            result(i + A.getRows(), j) = B(i, j);
        }
    }
    for (int i = 0; i < C.getRows(); ++i) {
        for (int j = 0; j < C.getColumns(); ++j) {
            result(i + A.getRows() + B.getRows(), j) = C(i, j);
        }
    }
    return result;
}

// Horizontal stack of three matrices
Matrix hstack(const Matrix &A, const Matrix &B, const Matrix &C) {
    if (A.getRows() != B.getRows() || A.getRows() != C.getRows()) {
        throw std::invalid_argument("hstack requires matrices of the same height.");
    }
    int w = A.getColumns() + B.getColumns() + C.getColumns();
    Matrix result(A.getRows(), w);
```

```
    for (int i = 0; i < A.getRows(); ++i) {
        for (int j = 0; j < A.getColumns(); ++j) {
            result(i, j) = A(i, j);
        }
    }
    for (int i = 0; i < B.getRows(); ++i) {
        for (int j = 0; j < B.getColumns(); ++j) {
            result(i, j + A.getColumns()) = B(i, j);
        }
    }
    for (int i = 0; i < C.getRows(); ++i) {
        for (int j = 0; j < C.getColumns(); ++j) {
            result(i, j + A.getColumns() + B.getColumns()) = C(i, j);
        }
    }
    return result;
}
```

3.1.5 Markowitz.cpp

```
//Markowitz.h
#ifndef MARKOWITZ_H
#define MARKOWITZ_H

#include "linalg.h"

#define TOLERANCE 1e-10
typedef enum {NODEBUG, DEBUG};

class Markowitz {
private:
    Matrix returns;
    Matrix target_returns;
    Matrix optimal_weights;
    int n;

public:
    Markowitz(const Matrix &_returns, const Matrix &_target_returns) {
        if (_target_returns.getRows() != 1) {
            throw std::invalid_argument("target returns should be a column vector");
        }
        returns = _returns; target_returns = _target_returns, n = _returns.getRows();
        this->weights();
    }

    // Mean returns
    Matrix mean();

    // Covariance matrix
    Matrix cov();

    // Q matrix
    Matrix Q();

    // b Vector
    Matrix b(const double &target_return);

    // weights dataframe
    Matrix weights();

    Matrix getWeights() {return this->optimal_weights;}

    void NormTest();

};

#endif

//Markowitz.cpp
```

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

// implement summation in the pdf
Matrix Markowitz::mean() {
    int assets = returns.getRows();
    int days = returns.getColumns();
    Matrix result(assets, 1);
    double mean;
    for (int i = 0; i < assets; i++) {
        mean = 0;
        for (int j = 0; j < days; j++)
            mean += returns(i, j);
        mean /= days;
        result.insert(i, 0, mean);
    }
    return result;
}

// implement summation in the pdf
Matrix Markowitz::cov() {
    int assets = returns.getRows();
    int days = returns.getColumns();
    Matrix rBar = this->mean();         // current portfolios mean()
    Matrix result(assets, assets);
    double sum;
    for (int i = 0; i < assets; i++) {
        for (int j = 0; j < assets; j++) {
            sum = 0;
            for (int k = 0; k < days; k++) {
                sum += (returns(i, k) - rBar(i, 0)) * (returns(j, k) - rBar(j, 0));
            }
            sum /= (days - 1);
            result.insert(i, j, sum);   // result[i][j] = sum. with check
        }
    }
    return result;
}

Matrix Markowitz::Q() {
    Matrix e(Lattice(returns.getRows(), Vector(1, 1.0)));       // ones column vector
    Matrix r = this->mean();
    Matrix zero(Lattice(1, Vector(1, 0.0)));                    // 1*1 Matrix containing 0.0
    Matrix top_row = hstack(this->cov(), -r, -e);
    Matrix middle_row = hstack(-r.transpose(), zero, zero);
    Matrix bottom_row = hstack(-e.transpose(), zero, zero);
    Matrix A = vstack(top_row, middle_row, bottom_row);
    return A;
}

Matrix Markowitz::b(const double &target_return) {
    Matrix zeros(Lattice(returns.getRows(), Vector(1, 0.0)));
    Matrix ret(Lattice(1, Vector(1, target_return)));           // 1*1 Matrix containing double
    Matrix one(Lattice(1, Vector(1, 1.0)));                     // 1*1 Matrix containing 1.0
    Matrix b = vstack(zeros, -ret, -one);                       // Unary negative required
    return b;
}

Matrix Markowitz::weights() {
    int m = target_returns.getColumns();
    Matrix results(m, this->n + 2);
    // optimal weights Matrix 21 * 85 (assets + mu and lambda)

    // For each target return
    for (int i = 0; i < m; i++) {
        Matrix x = this->Q().solver(this->b(target_returns(0, i)), TOLERANCE, NODEBUG);

        // insert the x vector into the optimal weights matrix
        for (int j = 0; j < this->n + 2; j++) {
```

```
            results.insert(i, j, x(j, 0));
        }
    }
    this->optimal_weights=results;
    return results;
}

void Markowitz::NormTest() {
    Matrix Q = this->Q();
    for (int i = 0; i < target_returns.getColumns(); i++) {
        double tr = target_returns(0,i);
        Matrix bx = b(tr);
        std::cout << "Norm for target return " << tr << ":   " << \
            (Q * this->optimal_weights(i,i+1,0,85).transpose() - bx).norm() << std::endl;
    }
}
```

3.1.6 write_data.cpp

```
//write_data.h
#ifndef WRITE_DATA_H
#define WRITE_DATA_H

#include <linalg.h>
#include <fstream>
#include <sstream>
#include <unistd.h>

struct Result {
    int id;
    Matrix back_test;
    Matrix weights;
};

// Write results array to a csv format
void write_data(const Result *results, size_t size);

// converts a Lattice to a string for write_data usage
std::string vectorToString(const Lattice& M);

#endif

//write_data.cpp
#include "write_data.h"

// Function to convert a vector of vectors to a string
std::string vectorToString(const Lattice& M) {
    std::ostringstream oss;
    for (const Vector& innerVec : M) {
        for (size_t i = 0; i < innerVec.size(); ++i) {
            oss << innerVec[i];
            if (i != innerVec.size() - 1) {
                oss << " "; // Space-separated values within inner vectors
            }
        }
        oss << ";"; // Semicolon-separated inner vectors
    }
    return oss.str();
}

void write_data(const Result *results, size_t size){
    // Print the current working directory
    char cwd[PATH_MAX];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        std::cout << "Current working directory: " << cwd << std::endl;
    } else {
        perror("getcwd() error");
    }
    // Create an ofstream object to write to a file
```

```
    std::ofstream outFile("output_data.csv");

    // Check if the file is open
    if (outFile.is_open()) {
    // Write the CSV header
    outFile << "ID,back_test,weights\n";

    // Write each data entry to the CSV file
    for (size_t i = 0; i < size; ++i) {
        const Result& data = results[i];
        outFile << data.id << "," << vectorToString(data.back_test.getMatrix()) << "," \
            << vectorToString(data.weights.getMatrix()) << "\n";
    }

    // Close the file
    outFile.close();
    std::cout << "Writing to CSV file completed successfully." << std::endl;
    } else {
        std::cerr << "Unable to open file for writing." << std::endl;
    }
}
```