Nikolaos Petsas

CID: 01904210

Computational Finance With C++

In this code I construct a portfolio optimization solver and perform backtesting to assess the performance of the Markowitz model.

The first action in the main function is to get the data from a csv file. For this action I need the csv and csv.h files that perform this job. The data are stored in a Matrix<double> which is a template function that I construct in the Matrix.h file. The main element of this class is the myMatrix variable which is a vector<vector<type> >.This structure is very useful for the rest of the code as I will need to construct matrices that sometimes has other matrices as elements. The class provides us with the functions SetMatrix(for initializing the sizes), Setters and Getters both for the whole matrix and for a specific element.

After that, I need the Q, xo and b matrices for optimization. All these matrices are in the form of Matrix<vector<vector<double> > > as they are matrices containing other matrices. To construct them, I set up all elements that are included.

$$Qxo=b \Rightarrow \begin{bmatrix} \Sigma & -r & -e \\ -r^T & 0 & 0 \\ -e^T & 0 & 0 \end{bmatrix} \begin{bmatrix} w \\ \lambda \\ \mu \end{bmatrix} = \begin{bmatrix} 0 \\ -r_p \\ -1 \end{bmatrix}$$

Where Σ is the covariance matrix(83x83) for all assets, r is the vector(83x1) with average returns for all assets, e is a vector with ones, w is the vector of weights, rp is the target return for the portfolio.

I set up all these matrices by using the Matrix class. I also set up matrices for Q, guess (the x0), b and b-Qxo.

After the initialization of matrices, the code inserts a for loop that runs from the first until the 20th portfolio. It starts performing the strategy. First of all, it calculates the average returns and covariance matrices by using the functions calculateReturns and calculateCovariance. Standard matrices which have the same element at all nodes are calculated in the calculateMatrix function before the for loop. All functions that fill some matrix are in the file FillMatrices.cpp part of the header file FillMatrices.h

The basic class that implements functions for calculating Q, b and b-Qxo is the Eq1 that inherits from abstract class Equation. Here, my initial purpose was to have a base class Equation that provides a virtual function that calculates the b-Qxo. I wanted this b-Qxo to be passed by class pointer into the optimization algorithm so that if we wanted another expression to be passed into the algorithm, we could construct another class that inherits from Equation. But, something went wrong with private variables. So, inheritance has not important reason in my code.

In Equation.cpp file I perform the calculations for Q, b and b-Qxo variables. For these calculations, I need additions, multiplications between matrices, scalar multiplication of matrices or transpose of matrices. C++ does not provide these operations so I constructed a new header file that performs them (Operations.h, Operations.cpp).

When I get the b-Qxo expression, I passed to the algorithm function of Conjugate class where the optimization takes place. As the original conjugate gradient method requires a symmetric matrix, I used the biconjugate gradient method where instead of $s_k{}^T$ I use $\hat{s}_k$ which is an arbitrary matrix that can be multiplied with s_k The method is described here:
https://en.wikipedia.org/wiki/Biconjugate_gradient_method

I used the unpreconditioned version of the algorithm.

For the backtesting of the strategy, I chose to get the cumulative returns of our portfolio which is defined as (1+r1)*(1+r2)*(1+r3)*… *(1+r_n) -1. So first, I calculate the cumulative return of each asset in the 12 days of our sample. I store them into out_of_sample_cum_returns. Then, I get the portfolio_return by multiplying the cumulative returns with the weights that I found in optimization. I store this return into the ret vector. I do the same for covariances and store the portfolio_std into the cov vector. When the algorithm exit the inner loop (end of rolling window), I store these two vectors into two other vectors of vectors (final_returns_vector, final_cov_vector). Finally, after I run the whole strategy 20 times with 20 different targets I write out two csv files. The first (returns) contains the cumulative returns of 20 portfolios for each one of the 50 periods that I backtested the strategy. The second contains the portfolios std.

In the returns file we see that all portfolios have positive average cumulative returns and as we increase the target return, we get a higher return in the backtesting (0.2096 for the portfolio with the highest target return). On the other hand, as we increase the target for the return, we get also a higher risk (0.000535 for 0% target, 0.050895 for 9.5% target). This is of course reasonable by the finance theory.

I have also the time series of cumulative returns of the two outlier portfolios.