MSc Financial Technology

BUSI97153 - Computational Finance with C++

Individual Coursework:

Portfolio Optimization Solver and Backtesting Markowitz Model

Alfred Choi - 02161574

I. Project Objective and Description

This computational finance project aims to develop a portfolio optimization solver that is built in C++, to perform comprehensive backtesting analysis of the Markowitz model. The Markowitz model, introduced by Harry Markowitz in 1952, provided a mathematical optimisation approach that minimizes the portfolio variance, subject to expected return targets and budget constraints, which is the foundation of modern portfolio theory.

With C++ software architecture, this empirical study examines 83 companies from the FTSE 100 index across 700 trading days of return data. The methodology involves a rolling window approach, with 100-day in-sample calibration for estimating minimum-variance portfolio weights, followed by 12-day out-of-sample performance evaluation period. Starting from the 100th day, the in-sample window rolls over every 12 days until the end of the dataset.

In the following sections, we will detail the algorithmic steps of the model construction, and present important findings regarding the performance and efficiency of this quantitative risk-based strategy.

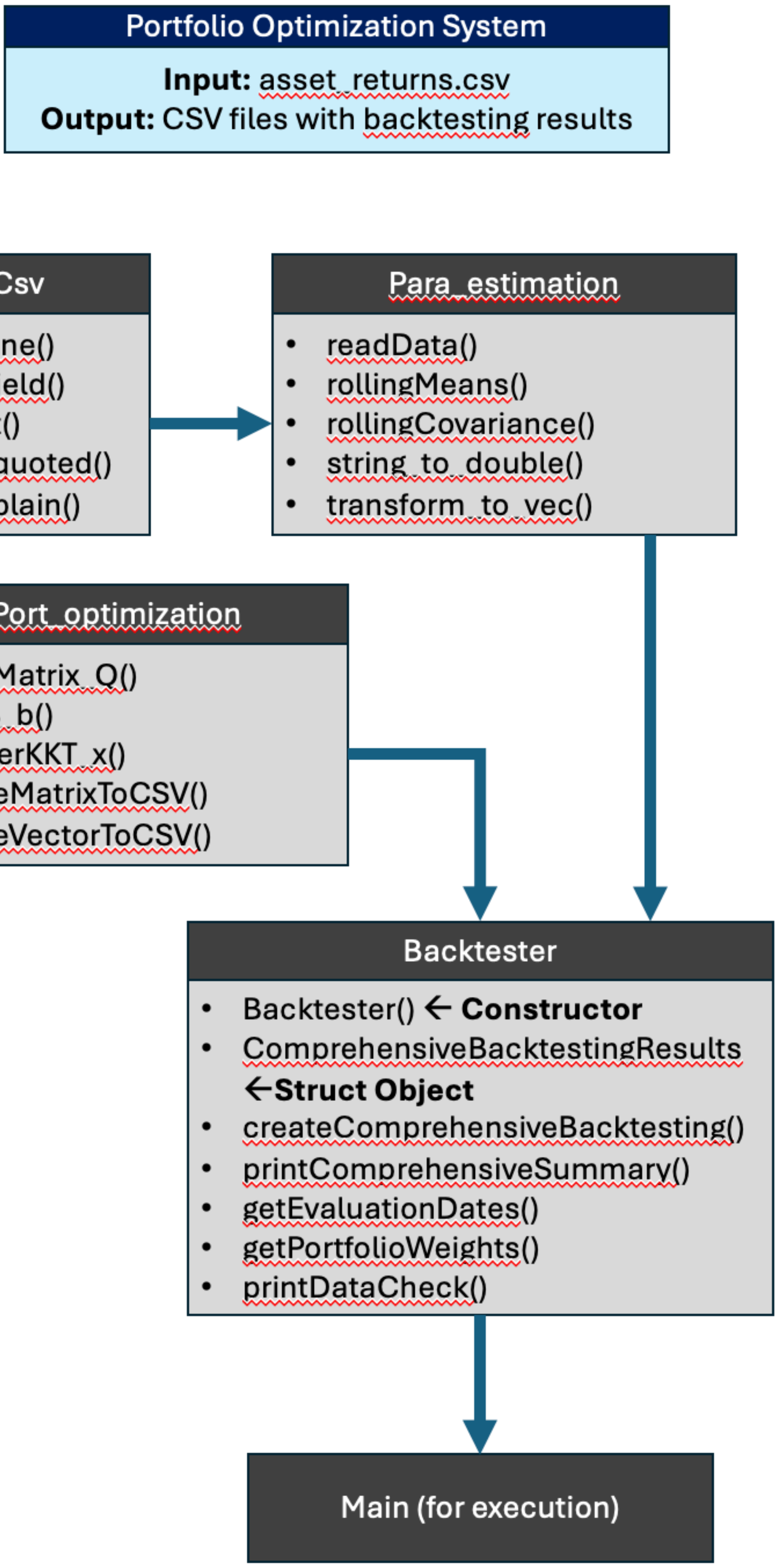
II. Software Structure

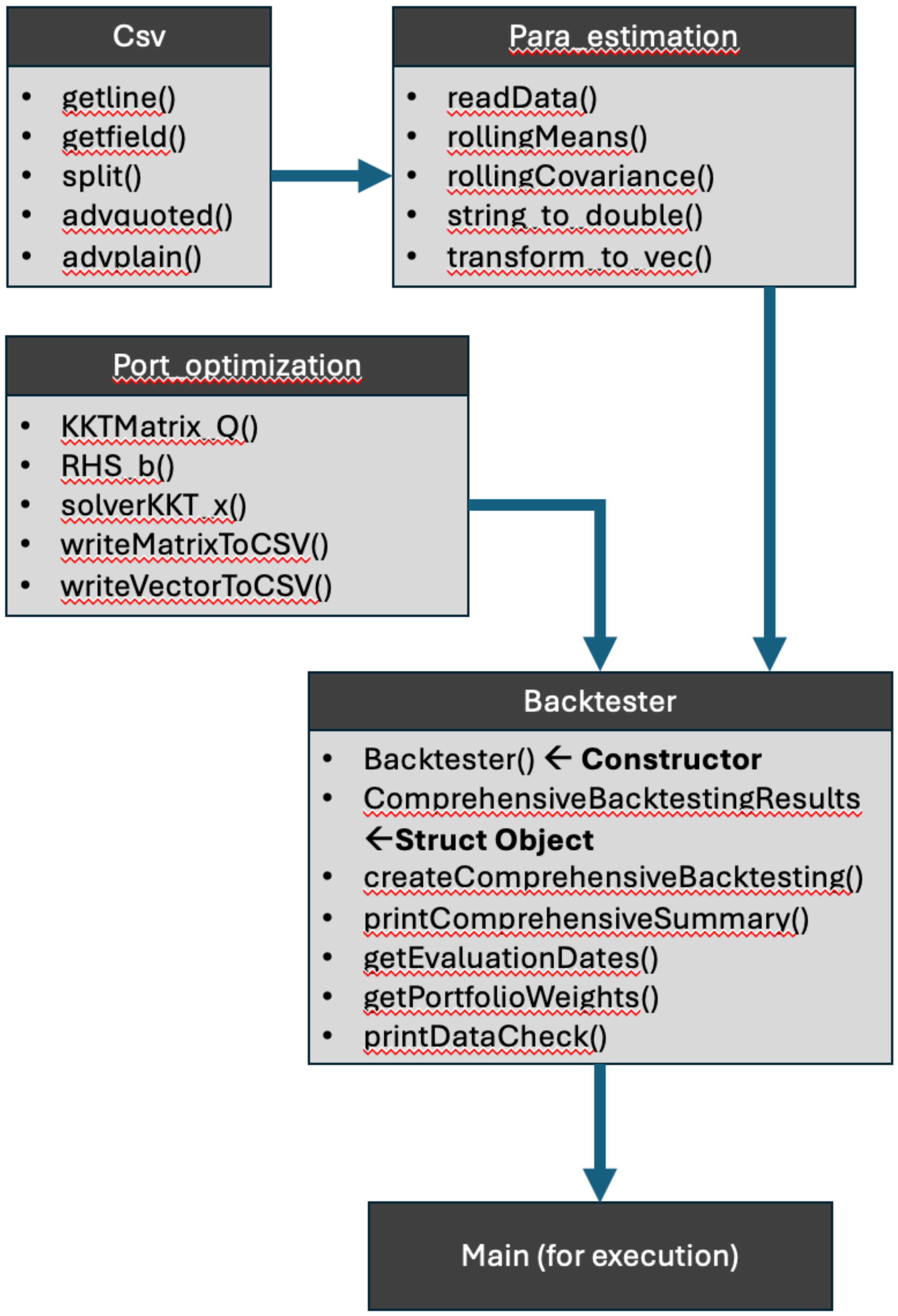
1. Class Structure & Linkage

The entire portfolio optimization software adopts an object-oriented architecture which consists of four linked classes:

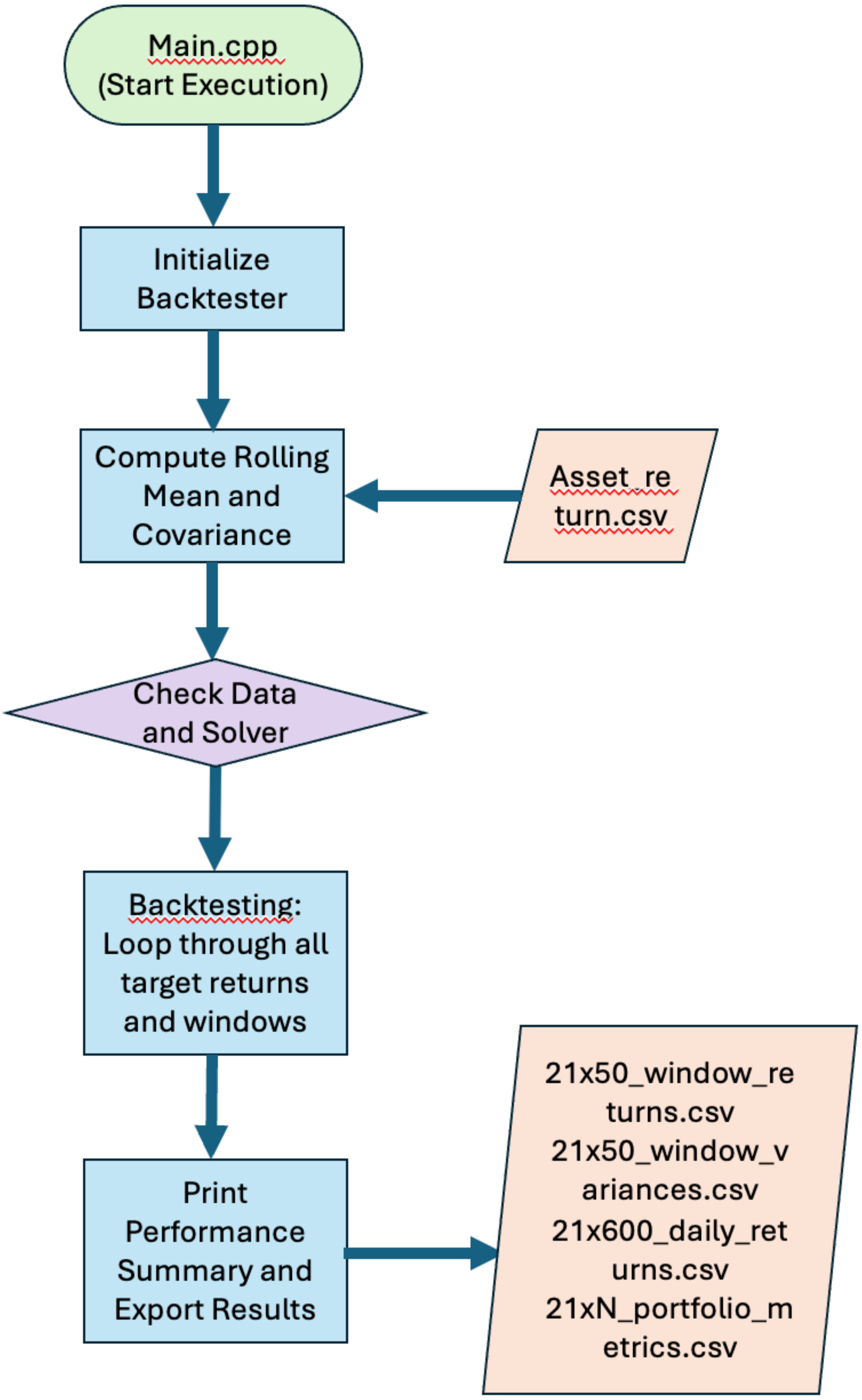
- **Csv:** This class is originally provided by the coursework. It contains several utility functions to read in and handle (.csv) comma-separated data files. The class firstly stores data in a dynamic pointer of pointer structure, before transforming into vector containers. Its functions are mainly called by the next function class *para_estimation*.
- **Para_Estimation**: This class serves as a statistical calculator, dealing with part (A) of the coursework requirement – parameter estimation. Using two core algorithms *rollingMeans()* and *rollingCovariance()*, the computer software calculates the time-series mean (83 companies x 700 periods matrix) and covariance (700 x 83 x 83 cube structure) for each day. The class also transforms raw pointer-based arrays into vector containers, using *transform_to_vec()* private functions.
- **Port_Optimization**: This class tackles the mathematical core of the coursework, part (B) – portfolio optimization. It performs the mean-variance optimization matrices through a Karush-Kuhn-Tucker (KKT) constrained framework. On the left, *KKTMatrix_Q()* creates a 85 x 85 matrix that incorporates the covariance for each time period, along with the target return and budget constraint. On the right, the *RHS_b()* function defines the constraint values in the last 2 rows of linear system. To solve the optimal point under the Lagrange constraints, *solverKKT_x()* carries out the Conjugate Gradient method, which iteratively refines the solution vector until convergence criteria is satisfied within the tolerance. The implementation and mathematical formulation will be detailed in the later section.
- **Backtester**: This class contains wrapper functions such as *computeRollingParameters()* and *createComprehensiveBacktesting()*, streamlining all workflows such as data loading, parameter setting, running optimization, and reporting results. This approach significantly simplifies and reduces the additional effort needed to link up all functions and variables during the execution in the main.cpp file. Moreover, this class builds a comprehensive object *Backtester*, covering a full suite of backtesting algorithms and metrics for coursework part (C). Users can thoroughly record all mean returns, variance, and Sharpe ratio - in a daily, window, annualized, and all-time bases.

1. Class Structure and Linkage





2. Main Programme Flow

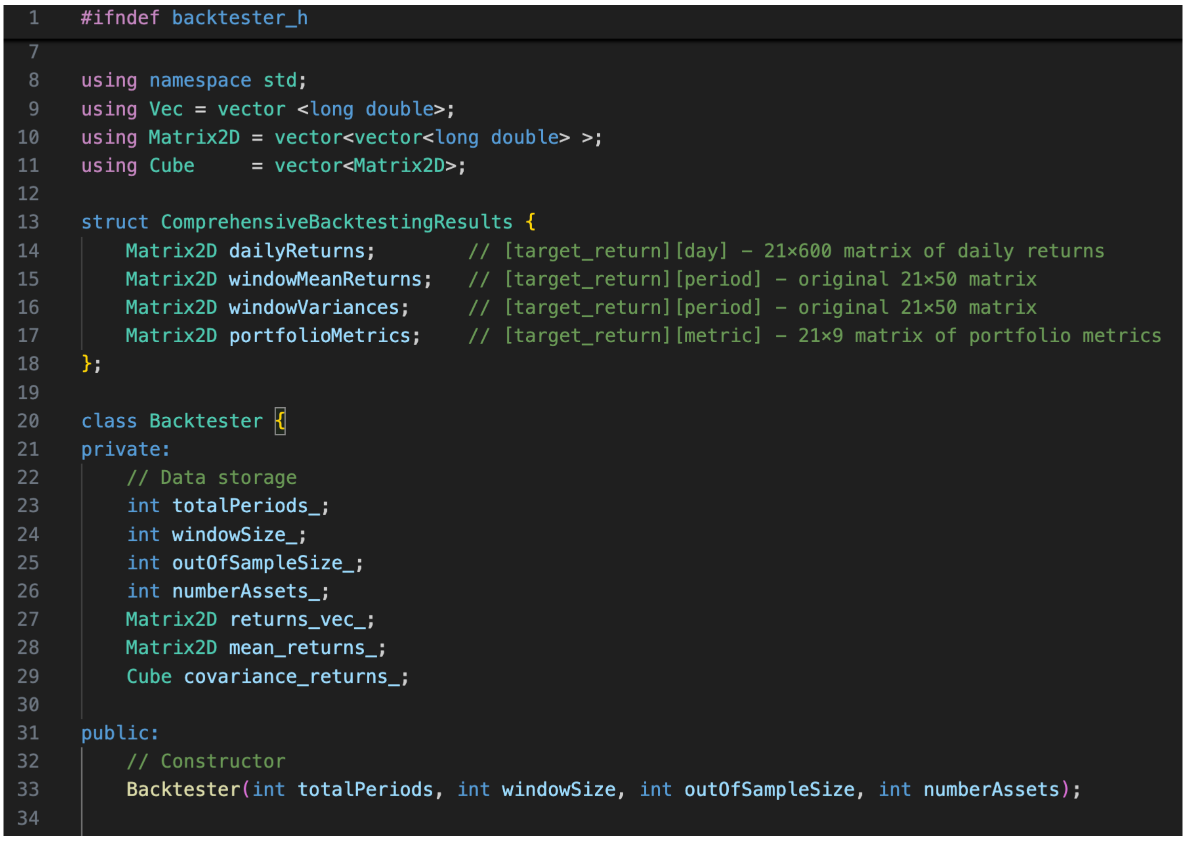

Main.cpp (Start Execution)
Initialize Backtester
Compute Rolling Mean and Covariance
Asset_return.csv
Check Data and Solver
Backtesting: Loop through all target returns and windows
Print Performance Summary and Export Results
21x50_window_returns.csv
21x50_window_variances.csv
21x600_daily_returns.csv
21xN_portfolio_metrics.csv

2. Main Programme Flow

While executing the computer software through *main.cpp*, the following stages are performed in a pipeline through subsequent function calls:

Stage 1: Initialize the Backtester

The *Backtester* object (from *backtester.h*) is initialized with predefined parameters (700 total periods, 100-day estimation window, 12-day evaluation period, 83 assets). This built a solid framework for further analysis, as all functions called in the main process are accessible through the object. This constructor also standardizes the essential parameter names (see below) used in the backtesting process:



```
#ifndef backtester_h

using namespace std;
using Vec = vector <long double>;
using Matrix2D = vector<vector<long double> >;
using Cube     = vector<Matrix2D>;

struct ComprehensiveBacktestingResults {
    Matrix2D dailyReturns;        // [target_return][day] - 21×600 matrix of daily returns
    Matrix2D windowMeanReturns;   // [target_return][period] - original 21×50 matrix
    Matrix2D windowVariances;     // [target_return][period] - original 21×50 matrix
    Matrix2D portfolioMetrics;    // [target_return][metric] - 21×9 matrix of portfolio metrics
};

class Backtester {
private:
    // Data storage
    int totalPeriods_;
    int windowSize_;
    int outOfSampleSize_;
    int numberAssets_;
    Matrix2D returns_vec_;
    Matrix2D mean_returns_;
    Cube covariance_returns_;

public:
    // Constructor
    Backtester(int totalPeriods, int windowSize, int outOfSampleSize, int numberAssets);

```

Stage 2: Compute Rolling Mean and Covariance

Calling *backtester.computeRollingParameters(),*the function further invokes four functions from the *ParameterEstimator* class (from *para_estimation.h*):

- *ParameterEstimator::readData(returnMatrix, fileName)*
  It reads in the asset_return.csv file that contains 83 company x 700 days return matrix, and stores the data in a 2-deimensional pointer structures (long double**).

- *ParameterEstimator::transform_to_vec(returnMatrix, numberAssets_, totalPeriods_)*
  It converts the pointer data structure into vector containers (Matrix2D), which is more efficient for matrix mathematical operation in later stages.

- *ParameterEstimator::rollingMeans(returns_vec_)*
  It calculates rolling mean returns using a 100-day sliding window, which produces 83 x 700 Matrix2D average return for each company in previous 100 days. Note that the first 99 days are marked as 0 due to insufficient historical data.

- *ParameterEstimator::rollingCovariance(returns_vec_, mean_returns_)*
  It calculates rolling covariance of returns using a 100-day sliding window, which produces 700 x 83 x 83 Cube covariance matrix using previous 100-day return data of all companies. Note that the first 99 days are marked as 0 due to insufficient historical data.

Stage 3: Check Data and Solver

Calling *backtester.printDataCheck()*,the function validates data through sampling a few mean return and covariance matrix data points to cross check against raw data, ensuring there is no discrepancy during the data transformation process.

It also executes *backtester.getPortfolioWeights(int period, long double targetReturn)* to test the validity and readiness of the portfolio optimization solver from the *PortfolioOptimizer* class (from *para_estimation.h*). See below for the KKT framework and methodology:

- *Matrix2D PortfolioOptimizer::KKTMatrix_Q(Sigma, meanReturn)*
  It constructs 85 x 85 KKT system matrix for constrained optimization, combining the covariance matrix Σ in the top-left block, - $\bar{r}$ (mean returns of past 100 days) in second last column and row, and -e negative unit vector of ones in the last column and row.

- *Vec PortfolioOptimizer::RHS_b(numberAssets_, targetReturn)*
  It constructs the right-side vector b of the linear system. The vector consists of zeros for the first 83 elements (numberAssets_), negative target return (return constraint) in the 84th row and -1 (budget constraint) in the 85th row.

- *Vec PortfolioOptimizer::solveKKT_x(Q, b)*
  It implements the **conjugate gradient method** to solve the linear system Qx = b iteratively.

$$\begin{pmatrix} \Sigma & -\bar{\boldsymbol{r}} & -\boldsymbol{e} \\ -\bar{\boldsymbol{r}}^\top & 0 & 0 \\ -\boldsymbol{e}^\top & 0 & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{w} \\ \lambda \\ \mu \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ -\bar{r}_{\mathrm{P}} \\ -1 \end{pmatrix}.$$

Where λ and μ are the Lagrange multipliers, and $\bar{r}_P$ is the target return, $\boldsymbol{e}$ represents a vector of ones.

```
Algorithm 1: Conjugate Gradient Method – Quadratic Programs
  Input : Initial point $x_0$, matrix $Q$, right hand side vector $b$, and solution
          tolerance $\epsilon$.
  0. Initialize:
          $s_0 = b - Qx_0, \; p_0 = s_0.$
  1. For $k = 0, 1, \ldots,$ until $s_k^\top s_k \le \epsilon$ do:
          $\alpha_k = \frac{s_k^\top s_k}{p_k^T Q p_k}$
          $x_{k+1} = x_k + \alpha_k p_k$
          $s_{k+1} = s_k - \alpha_k Q p_k$
          $\beta_k = \frac{s_{k+1}^\top s_{k+1}}{s_k^\top s_k}$
          $p_{k+1} = s_{k+1} + \beta_k p_k$
```

Denote the system of linear equations as $Qx = b$. Then the following algorithm (called the conjugate gradient method) can be used to solve it.

The algorithm follows closely with the above approach provided by the coursework:

- **Initialization:** It makes initial guess x0, with equal portfolio weights (1/numberAssets) and reasonable Lagrange multipliers (λ = target return, μ = 1.0). It also computes initial residual parameter r = b - Q*x (r is equivalent as parameter s in the coursework), and set the initial search direction p = r.

- **Convergent Gradient Loop Parameters:**
  - **Step size: $\alpha_k = r_k^T r_k/(p_k^T Q p_k)$**
    It is computed as the ratio of current residual squared to the quadratic form of search direction.
  - **Solution update: $x_{k+1} = x_k + \alpha_k p_k$**
    The new weight solution is updated by moving from its original point $\mathbf{x_k}$ along the search direction scaled by the optimal step size $\boldsymbol{\alpha}$.
  - **Residual update: $r_{k+1} = r_k - \alpha_k Q p_k$**
    Using the same logic, the residual is updated by subtracting the matrix-vector product of Qp_k scaled by optimal step size α from the current residual $\mathbf{r_k}$.
  - **Conjugate direction parameter: $\beta_k = r_{k+1}^T r_{k+1}/r_k^T r_k$**
    The beta coefficient calculates how much memory should be kept from the previous search direction for next move. This mixing ratio is calculated by the ratio between updated residual square to the old residual square.
  - **New search direction: $p_{k+1} = r_{k+1} + \beta_k p_k$**
    This combines current residual vector (steepest descent direction) with a beta ratio of previous search direction.

- **Convergence Control:** The computation adopts a stricter tolerance at 1e-12. The algorithm stops when the residual square is lower than the tolerance, or maximum iteration is reached (2 times the number of assets)

- **Matrix Operations Private Functions:** The solver also uses helper function that is designed in *PortfolioOptimize*r private class members, including:
  - *dot(const Vec &a, const Vec &b)* for vector dot products,
  - *MatM_Vec(Q, x)* for matrix-vector multiplication, and
  - *alpha_x_plus_y(long double alpha, const Vec &x, Vec &y)* for vector addition.

Stage 4: Backtesting Loop

This stage serves as the core computation engine that comprehensively backtests through the nested loops, iterating across 21 target return levels (0% to 10% in 0.5% increments) and 50 evaluation windows (12 days per window).

Firstly, the algorithm builds a structured data container - *ComprehensiveBacktestingResults results* – which organizes all backtesting outputs in the following components:

- results.dailyReturns (21 target returns x 600 daily returns)
- results.windowMeanReturns (21 target returns x 50 windows mean return)
- results.windowVariances (21 target returns x 50 windows return variance)
- results.portfolioMetrics (21 target returns x 9 performance metrics)

For each iteration, the system calls *Backtester::getPortfolioWeights(period, targetReturn)*, which is explained in stage 3, to solve the optimization problem. It then calculates the 12-day out-of-sample portfolio returns using the optimized weights.

Afterwards, the algorithm records the daily return, calculates returns mean and variance accordingly. After all periods return data is populated, the algorithm lastly evaluates the overall performance with different metrics and time frame across all target returns.

Stage 5: Print Performance Summary and Export Results

This stage finalizes the performance analysis with file outputs for users examination. Through calling the function *writeMatrixToCSV(const Matrix2D& matrix, const string& filename)*, the following csv files are exported:

- 21x50_window_returns.csv
- 21x50_window_variances.csv
- 21x600_daily_returns.csv
- 21xN_portfolio_metrics.csv
- portfolio_metrics_header.txt (explain column definitions for the portfolio metrics csv file)

Lastly, the system calls the return summary function *backtester.printComprehensiveSummary(const ComprehensiveBacktestingResults & results, long double riskFreeRate = 0.02L)*, which generates a formatted output in the terminal concluding the key performance metrics in a formatted way. This provides immediate insights about the performances and which level of the optimal target return gives the best results.

III. Performance Evaluation

Table 1: Portfolio Performance Summary Across Target Return Strategies (600-Day Evaluation)

Target Return	Daily Mean Return	Daily Variance	Daily S.D.	Daily Sharpe Ratio	Annual Variance	Annual S.D.	600-Day Cum Return	Annual Ret (from Cum)	Annual Sharpe Ratio
0.0%	0.14%	0.12%	3.45%	0.0396	29.9%	54.7%	64.80%	23.3%	0.3901
0.5%	0.20%	0.15%	3.83%	0.0522	37.0%	60.9%	111.7%	37.0%	0.6085
1.0%	0.22%	0.29%	5.34%	0.0406	71.9%	84.8%	52.9%	19.5%	0.2303
1.5%	0.25%	0.53%	7.27%	0.0343	133.3%	115.5%	-13.6%	-5.9%	N/A
2.0%	0.27%	0.88%	9.38%	0.0290	221.5%	148.8%	-68.3%	-38.3%	N/A
2.5%	0.31%	1.34%	11.59%	0.0265	338.6%	184.0%	-92.1%	-65.5%	N/A
3.0%	0.34%	1.91%	13.82%	0.0244	481.4%	219.4%	-98.9%	-85.1%	N/A
3.5%	0.36%	2.58%	16.07%	0.0223	651.0%	255.1%	-99.97%	-96.5%	N/A
4.0%	0.38%	3.36%	18.33%	0.0208	846.3%	290.9%	-100.0%	-100.0%	N/A
4.5%	0.41%	4.25%	20.62%	0.0200	1071.2%	327.3%	-100.0%	-100.0%	N/A
5.0%	0.44%	5.25%	22.91%	0.0191	1322.6%	363.7%	-100.0%	-100.0%	N/A
5.5%	0.47%	6.35%	25.20%	0.0187	1600.8%	400.1%	-100.0%	-100.0%	N/A
6.0%	0.49%	7.56%	27.50%	0.0180	1905.2%	436.5%	-100.0%	-100.0%	N/A
6.5%	0.51%	8.88%	29.79%	0.0173	2236.5%	472.9%	-100.0%	-100.0%	N/A
7.0%	0.55%	10.31%	32.11%	0.0173	2598.6%	509.8%	-100.0%	-100.0%	N/A
7.5%	0.56%	11.79%	34.34%	0.0163	2971.8%	545.1%	-100.0%	-100.0%	N/A
8.0%	0.66%	13.48%	36.72%	0.0179	3398.2%	582.9%	-100.0%	-100.0%	N/A
8.5%	0.63%	15.22%	39.01%	0.0161	3835.5%	619.3%	-100.0%	-100.0%	N/A
9.0%	0.64%	16.90%	41.11%	0.0157	4259.5%	652.6%	-100.0%	-100.0%	N/A
9.5%	0.69%	19.03%	43.63%	0.0159	4796.3%	692.6%	-100.0%	-100.0%	N/A
10.0%	0.98%	21.05%	45.88%	0.0214	5305.5%	728.4%	-100.0%	-100.0%	N/A

Table 1 presents the overall portfolio performance results by running the iterative computational backtesting analysis, covering 600 out-of-sample days with 12-day rebalancing intervals, under the assumptions of no short-selling constraints and zero transaction costs. Here are the key findings and discussions:

- **Optimal Target Return:** In the long-term perspective, setting the **<u>0.5% daily target return</u>** *(highlighted in yellow)* level gives the **<u>best risk-adjusted performance</u>**, generating 111.7% cumulative return (annualized 37.0%) over the 600-day rolling back-testing period. The Sharpe ratio is 0.6085 annually and 0.052 daily, which stands out from other groups.

  Note that in the long run returns compound differently, therefore annualized performance should be based on all-period return, rather than multiplying daily returns.

- **Performance Deterioration:** Although higher target return groups seemingly give higher out-of-sample daily return, their daily standard deviation also escalates considerably, which destroy their long-term return. Large losses create disproportionate recover requirements: for example, losing 50% requires a 100% gain to break even, while losing 90% needs a 900% gain. If the daily standard deviation is too high, **portfolios face high probability of irrecoverable loss**. This explains why **4.0% target daily return or above results in total investment loss** in the 600-day period.

- **Risk-Return Trade-off:** From the above table, higher target return group may bring higher daily mean return, but also leads to disproportionate increase in risk in terms of standard deviation. Therefore, we can see Sharpe ratio drops drastically for more aggressive return target, e.g. from 0.6085 at 0.5% target, to only 0.2303 at 1.0% target. This indicates diminishing efficiency of portfolio optimization and risk diversification if the target return is too aggressive.

- **Target vs Realized Return Constraints:** Despite the target daily return can be up to 10%, the realized daily mean return still stays below 1% across all groups. This shows that portfolio optimization cannot generate exceptional returns beyond the asset universe capabilities. Even with unlimited short selling, aggressive the return targets primarily escalate investment risks rather than improves the actual return performance. Investors therefore should set a **realistic investment objective that aligns with market realities**. For future analysis, it is suggested the portfolio optimization should focus on studying between 0% to 1% daily return range to find out a more precise target that maximizes risk-adjusted performance in the achievable market reality.

- **Leverage Effect and Short Selling:** Higher target returns require increased level of leverage and short selling. While the model permits these strategies to pursue higher mean return, the aggressive positioning also amplifies the extreme negative outcomes in adverse market conditions. As discussed, excessive leveraging and short selling negatively impact long-term performance due to compounding effect of risk and the asymmetric nature of recovery from large losses. It is also suspected whether brokerage accepts ultra-high leverage/ shorting selling level for their clients.

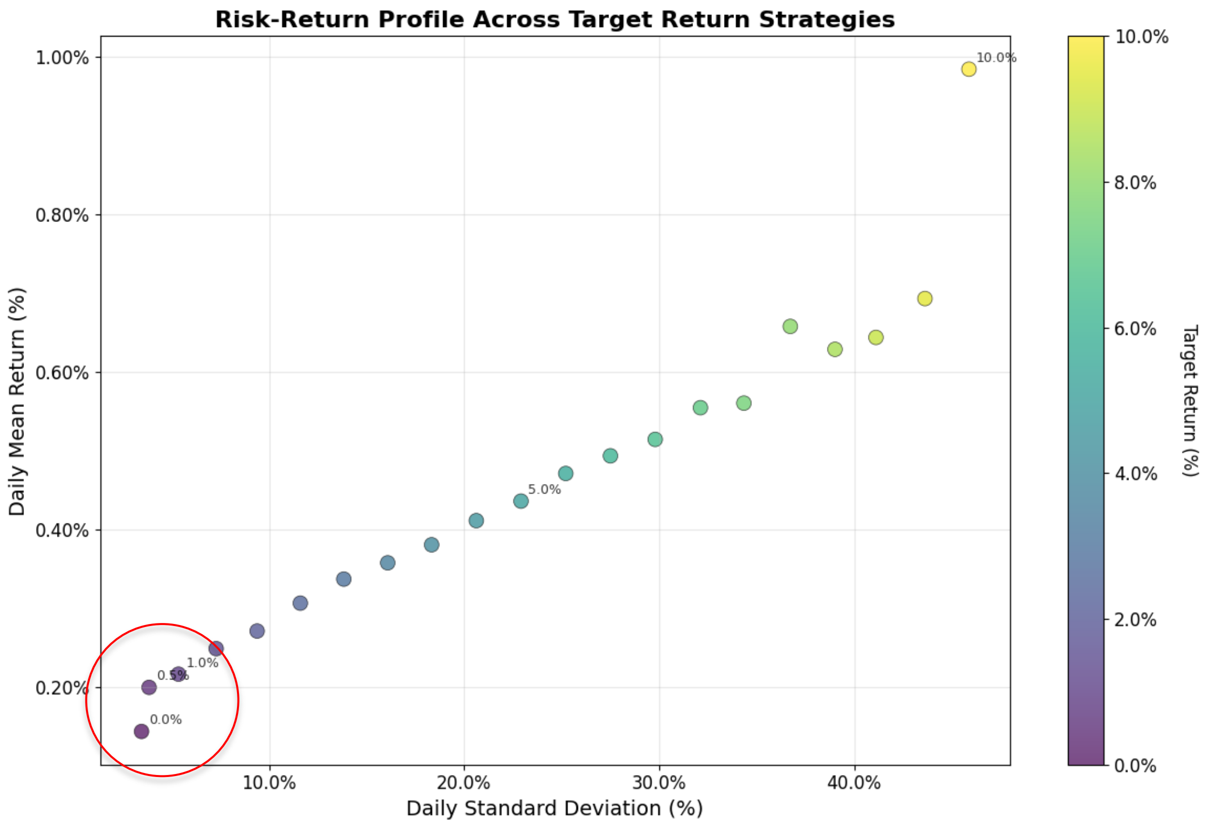
- **No Transaction Costs and Margin Requirements:** The results in Table 1 only reflects the theoretical performance without transaction costs or margin requirements. Therefore, it only shows the upper bound on achievable performance which would be much lower in real-world implementation. Transaction costs include bid-ask spreads, brokerage commissions, and market impact costs, which accumulate significantly under the frequent rebalancing across the 83 assets. Moreover, aggressive short-selling and leverage face substantial margin requirements and borrowing costs (short positions). Investors may also face forced liquidation during margin calls.

  These friction costs will exacerbate losses in high target return portfolios, while diminish gains for conservative target during rebalancing. Therefore, the real-world performance in execution may be significantly different from what we see in the above table.

Supplementary Diagrams for Performance Analysis

To enhance understanding of the portfolio strategy risk-return characteristics, this section provides supplementary diagrams and detailed analysis of the performance patterns.

Diagram 1: Risk-Return Profile Across Target Daily Returns



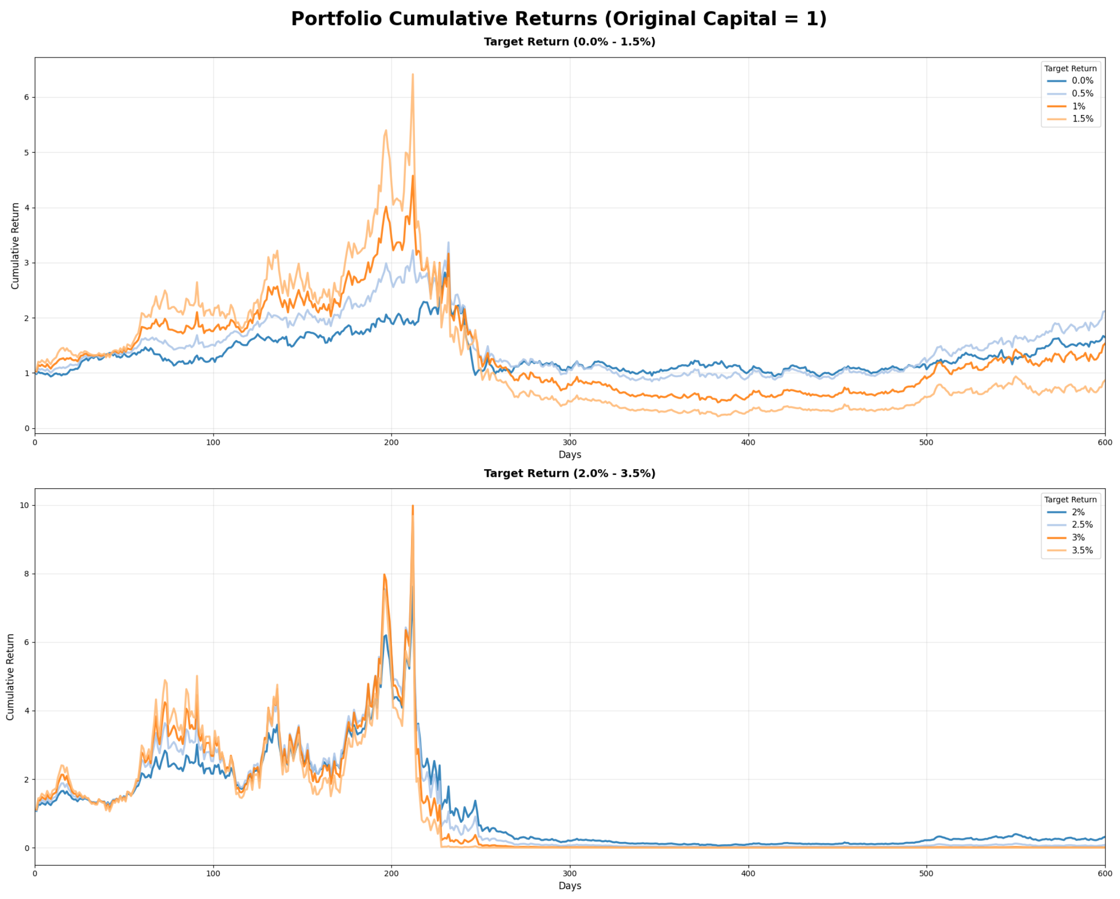
Findings and Discussions:

- **Efficient Frontier Identification:** The chart shows a concave curvature at lower left between the range of 0% and 1% of daily target return. This suggests there is steeper actual mean return enhancement with modest move of standard deviation. The portfolio risk-return characteristic is improved within this range.

- **Risk Escalation Beyond Optimal Range:** Beyond the 1% threshold, the increase in return becomes flatter compared to the increase of standard deviation. This pattern shows disproportionate volatility increase that substantially affect the risk-adjusted return in the long term.

- **Recommendation:** Further investigation should focus on the range between 0% and 1% daily target return to identify the best target return level, and the subsequent optimized portfolio weights for the portfolio.

Diagram 2: Portfolio Cumulative Returns over 600 days for different target return levels



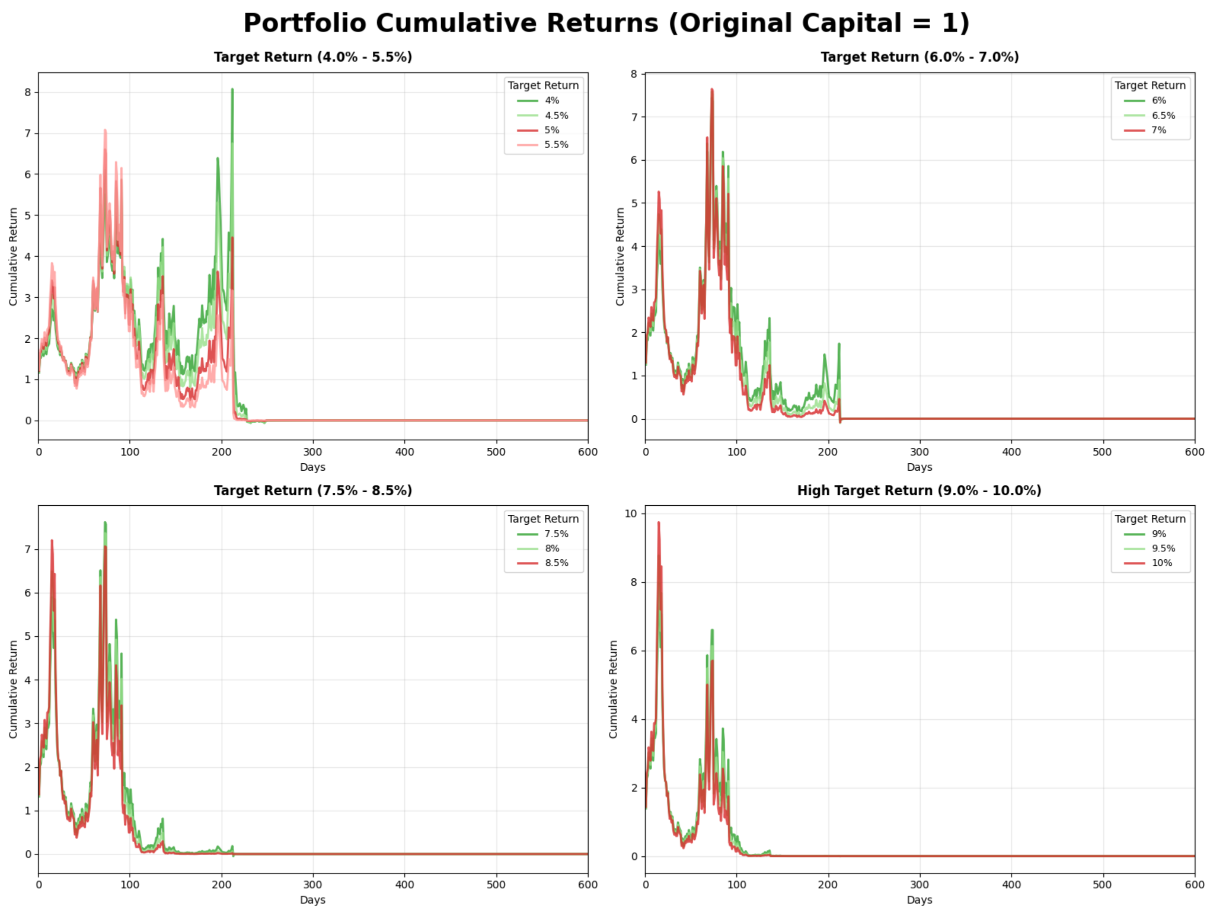
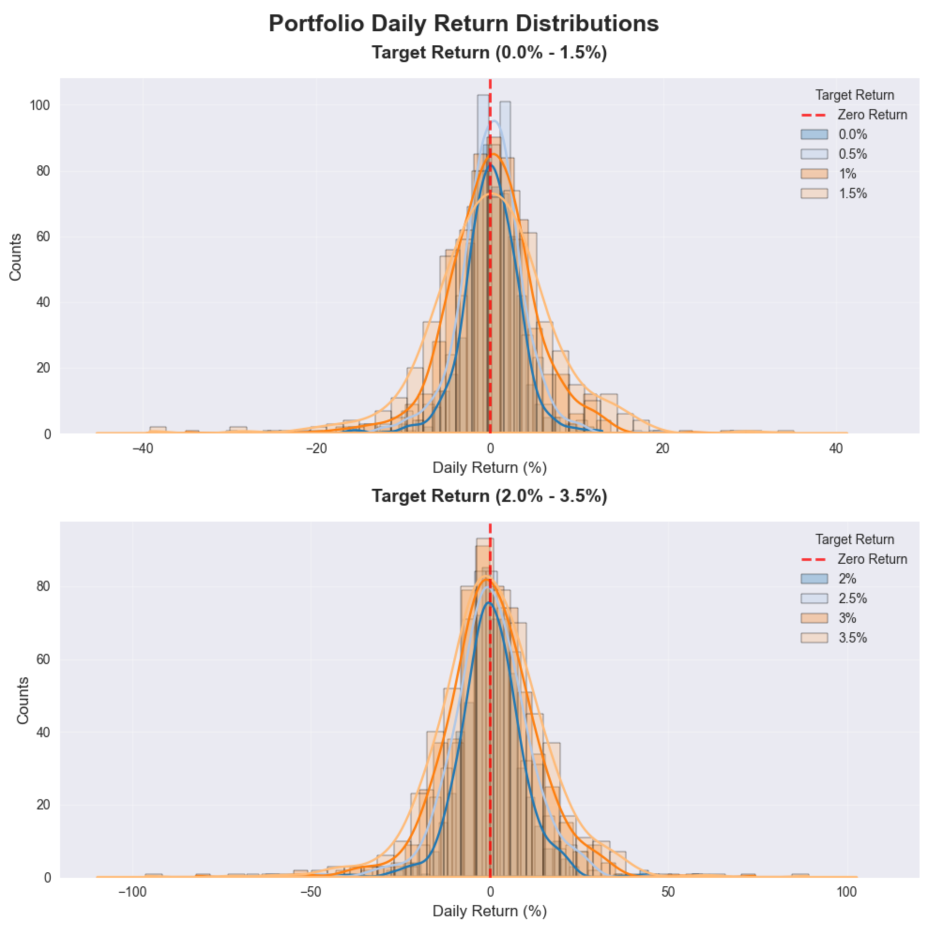
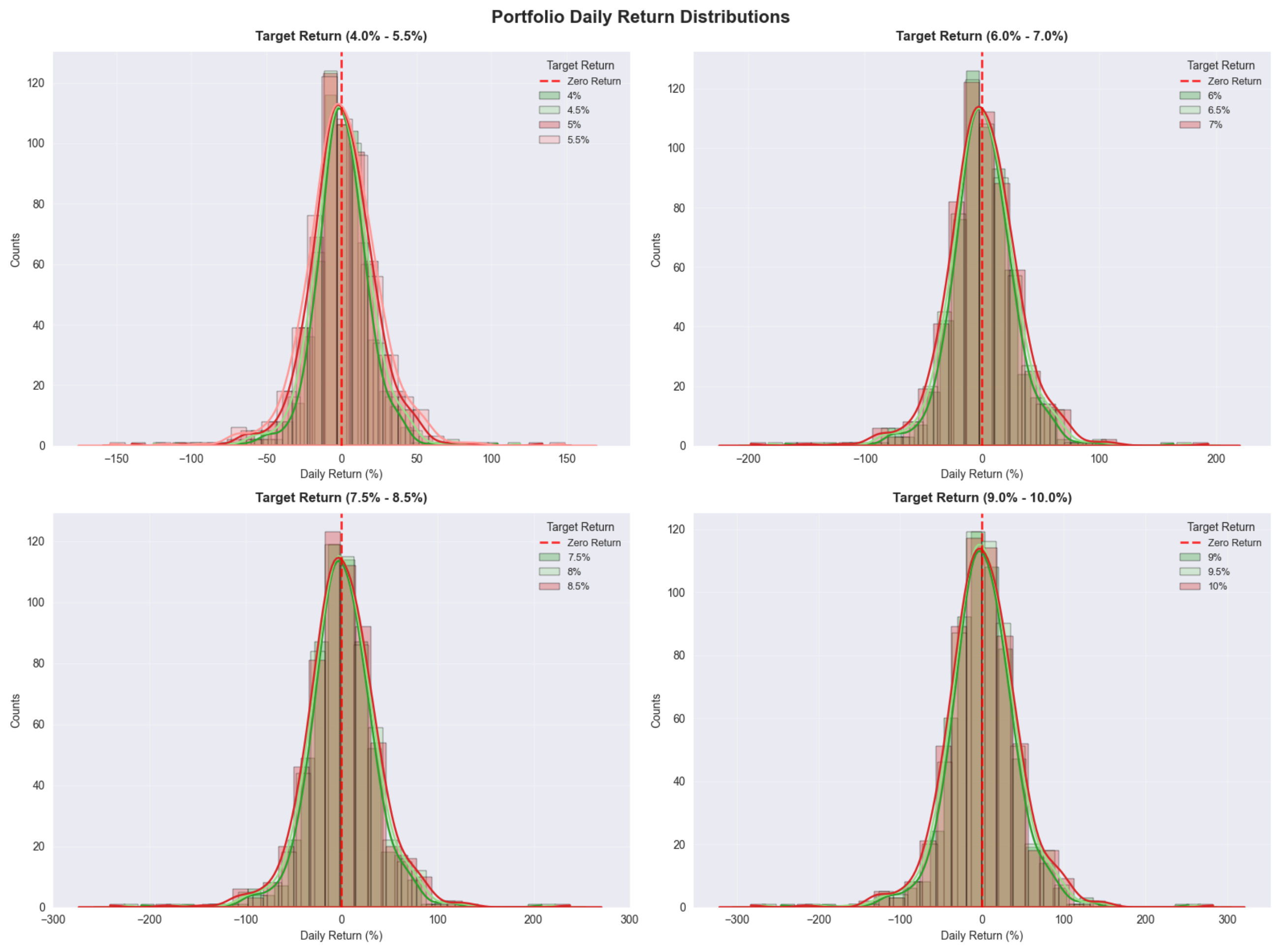


Diagram 3: Portfolio Cumulative Returns over 600 days for different target return levels





Findings and Discussions:

- **Bubble and Crash Pattern:** According to diagram 2, all portfolios experienced dramatic rise in return between days 100-200, spiking to the all-time-high over the whole investigation period. However, when the market conditions worsen, all portfolios face severe crashes. This suggests that the optimized portfolio is highly subject to speculating market-wide movements, especially when more leverage and short-selling is done for portfolios with high target returns.

- **Irreversible Capital Loss:** According to diagram 2, portfolios with target return exceeding 2% not only failed to recover from the original invested capital (below 1.0), but most of them also faced (near) total loss by the end of the sample date. This shows that aggressive return target poses a destructive result because daily volatility compounds much faster in downtrend than uptrend. This may lead to permanent loss rather than intended higher returns.

- **Normal Distribution with Amplified Tail Risk:** According to diagram 3, all portfolios' actual daily returns appear to follow normal distribution and centre around zero. However, higher target returns portfolios display significantly wider distributions, which corresponds to the elevated volatility as target return increases shown in Table 1.

  The most concerning area is that the significant left tail extension observed from the aggressive portfolios. It is not uncommon for their single day loss to exceed 50% or even 100%! These extreme levels represent devasting losses can happen within just a few days, yet are sufficient to destroy the entire portfolio holdings.

  Looking in detail, some bars on the left exceed the normal distribution curve. This indicates that while daily returns may appear statistically normal, extreme negative events become disproportionately likely. Investors should be aware the fat tail on the left – there is in a fact higher probability of experiencing rare but massive losses than they originally expected. The loss situation may worsen significantly if investors engage in high levels of leverage and short-selling activities.

IV. Appendix: C++ Computer Software Codes

main.cpp

```cpp
// main.cpp
#include "backtester.h"
#include <iostream>
#include <vector>

using namespace std;
using Vec = vector<long double>;

int main() {
    // Create backtester instance
    Backtester backtester(700, 100, 12, 83);

    // Compute rolling parameters
    backtester.computeRollingParameters();

    // Print data checks using the utility function
    backtester.printDataCheck();

    // Get evaluation dates and print them
    vector<int> evaluationDates = backtester.getEvaluationDates();

    // Sample Test - getting a point of portfolio weights for reference
    Vec weights = backtester.getPortfolioWeights(99, 0.09L);

    // Run comprehensive backtesting
    cout << "\n\nProcessing comprehensive backtesting with re-weighting in
every 12-day window..............." << endl;
    ComprehensiveBacktestingResults comprehensiveResults =
backtester.createComprehensiveBacktesting(0.02L);

    // Print the summary
    backtester.printComprehensiveSummary(comprehensiveResults, 0.02L);

    return 0;
}
```

para_estimation.h

```
#ifndef para_estimation_h
#define para_estimation_h

#include <iostream>
#include <vector>
#include <string>
#include <cmath>
using namespace std;
using Matrix2D = vector<vector<long double> >;
using Cube     = vector<Matrix2D>;


class ParameterEstimator {
public:
    // Constructor
    ParameterEstimator(int windowSize = 100);

    // Main functionality - Static utility functions
    static Matrix2D rollingMeans(const Matrix2D& returns, int windowSize =
100);
    static Cube rollingCovariance(const Matrix2D& returns, const Matrix2D&
meanreturns, int windowSize = 100);
    static double string_to_double(const string& s);
    static void readData(long double** data, const string& fileName);

    // Template method for data transformation
    template <class T>
    static vector<vector<T> > transform_to_vec(T** data, int rows, int
cols) {
        vector<vector<T> > data_vec(rows, vector<T>(cols));
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                data_vec[i][j] = data[i][j];
            }
        }
        return data_vec;
    }

private:
    // Private member variable, if any
    int windowSize_; //placeholder
};

#endif
```

para_estimation.cpp

```cpp
#include <iostream>
#include <cmath>
#include <vector>
#include "para_estimation.h"
#include "csv.h"
#include <stdio.h>
#include <fstream>
#include <stdlib.h>
#include <sstream>

using namespace std;
using Matrix2D = vector<vector<long double> >;
using Cube     = vector<Matrix2D>;


Matrix2D ParameterEstimator::rollingMeans (const vector<vector<long double> >
&returns, int windowSize){
    cout << "Calculating rolling means with window size " << windowSize << "..." <<
endl;
    int row_comps = returns.size();
    int col_periods = returns[0].size();
    vector <vector <long double> > return_means(
        row_comps, //no. of rows
        vector<long double> (col_periods, 0.0) //for each row, stroe a vector with
length col_periods, each filled with 0
    );

    for (int i = 0; i< row_comps; ++i){
        for (int j = 0; j<col_periods; ++j){
            if (j >= windowSize -1){
                long double sum = 0;
                for (int k = j - (windowSize - 1); k <= j; ++k){
                    sum += returns[i][k];
                }
                return_means[i][j] = sum/windowSize;
            }
        }
    }
    cout << "\nFinish running rollingMeans ....\n";
    return return_means;
};

Cube ParameterEstimator::rollingCovariance(
    const Matrix2D &returns, const Matrix2D &meanreturns,int windowSize){
        cout << "Calculating rolling covariance with window size " << windowSize <<
"..." << endl;
        int comps = returns.size();
        int periods = returns[0].size();
        Cube covariance_cube (periods, // rows
            Matrix2D(comps, vector <long double> (comps, 0.0))); //column becomes a
83 x 83 covariance matrix
```

```
        for (int p = windowSize - 1; p < periods; ++p){ //calendar index
            for (int comp1 = 0; comp1 < comps; ++comp1){
                for (int comp2 = 0; comp2< comps; ++comp2){
                    long double sum = 0;
                    for (int t = p - windowSize + 1; t<=p; ++t){
                        sum += ((returns[comp1][t] - meanreturns[comp1][p])*
(returns[comp2][t] - meanreturns[comp2][p]));
                    }
                    covariance_cube[p][comp1][comp2] = sum / (windowSize -1);
                }
            }
        }
        cout << "\nFinish running rollingCovariance ....\n";
        return covariance_cube;
    };

double ParameterEstimator::string_to_double(const string& s )
{
    istringstream i(s);
    double x;
    if (!(i >> x))
        return 0;
    return x;
}

void ParameterEstimator::readData(long double **data,const 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++)
            {
               long double temp=string_to_double(csv.getfield(j));
               cout << "Asset " << j << ", Return "<<i<<"="<< temp<<"\n"; //test
               data[j][i]=temp;
            }
            i++;
        }
        file.close();
    }
    else {cout <<fileName <<" missing\n";exit(0);}

    cout<<"Finish readData processing ----------";
}
```

port_optimization.h

```cpp
#ifndef port_optimization_h
#define port_optimization_h

#include <iostream>
#include <cmath>
#include <vector>

using namespace std;
using Vec = vector <long double>;
using Matrix2D = vector<vector<long double> >;
using Cube     = vector<Matrix2D>;

class PortfolioOptimizer {
public:
    // Main public functions
    static Matrix2D KKTMatrix_Q(const Matrix2D &Sigma, const Vec &meanreturn);
    static Vec RHS_b(int N, long double targetreturn);
    static Vec solveKKT_x(const Matrix2D &Q, const Vec &b);
    static void writeMatrixToCSV(const Matrix2D& matrix, const string& filename);
    static void writeVectorToCSV(const Vec& vector, const string& filename);

private:
    // Private helper functions
    static long double dot(const Vec &a, const Vec &b) {
        long double sum = 0.0L;
        for (int i = 0; i < a.size(); ++i) {
            sum += a[i] * b[i];
        }
        return sum;
    }

    static Vec MatM_Vec(const Matrix2D &A, const Vec &x) {
        int n = A.size();
        int m = x.size();
        Vec b(n, 0.0L);
        for (int i = 0; i < n; ++i) {
            long double sum = 0.0L;
            for (int j = 0; j < m; ++j) {
                sum += A[i][j] * x[j];
            }
            b[i] = sum;
        }
        return b;
    }
        static void alpha_x_plus_y(long double alpha, const Vec &x, Vec &y) {
        for (int i = 0; i < x.size(); ++i) {
            y[i] += alpha * x[i];
        }
    }
};
#endif
```

port_optimization.cpp

```cpp
#include <iostream>
#include <cmath>
#include <vector>
#include "para_estimation.h"
#include "port_optimization.h"
#include "csv.h"
#include <stdio.h>
#include <fstream>
#include <stdlib.h>
#include <sstream>
#include <numeric>

using namespace std;
using Vec = vector <long double>;
using Matrix2D = vector<vector<long double> >;
using Cube     = vector<Matrix2D>;


Matrix2D PortfolioOptimizer::KKTMatrix_Q(const Matrix2D &Sigma, const Vec
&meanreturn){
    int N = Sigma.size(); //find the number of assets
    int M = N+2;

    cout << "Checkpoint step 0: Algorithm has entered the function ......" << endl;
    cout << "Start Step 1 ......" << endl;

    //Step 1: Create a Zero Matrix
    Matrix2D Q(M, Vec(M,0.0L));

    cout << "\nCheckpoint step 1: Zero Matrix is created ......" << endl;
    cout << "********* Start Step 2 ...... ********* " << endl;

    //Step 2: Fill up the non-zero blocks in the matrix

    cout << "Start Top Left Sigma ......" << endl;
    //Top left sigma
    for (int i = 0; i < N; ++i){
        for (int j=0; j<N; ++j){
            Q[i][j] = Sigma[i][j];
        }
    }
    cout << "Finish Top Left Sigma ......" << endl;
    cout << "\nStart Top Right (meanreturn and e) ......" << endl;

    //Top right: -meanreturn and -e
    for (int i = 0; i < N; ++i){
        Q[i][N] = -meanreturn[i];
        Q[i][N+1] = -1.0L;
    }
    cout << "\nFinish Top Right (meanreturn and e) ......" << endl;

    cout << "Start Bottom Left  Transpose of (meanreturn and e) ......" << endl;
    //Bottom left: transpose of -meanreturn and -e
    for (int j = 0; j < N; ++j){
        Q[N][j] = -meanreturn[j];
        Q[N+1][j] = -1.0L;
    }
    cout << "Finish Bottom Left  Transpose of (meanreturn and e) ......" << endl;

    //Bottom right 2x2 space are still zeros
    cout << "Ready to return Matrix2D 'Q' " << endl;
    return Q;
};


Vec PortfolioOptimizer::RHS_b(int N, long double targetreturn){
    Vec b(N+2, 0.0L);
    b[N] = -targetreturn;
    b[N+1] = -1.0L;
    return b;
};

//improved version
Vec PortfolioOptimizer::solveKKT_x(const Matrix2D &Q, const Vec &b) {
    cout << "Start processing improved solverKKT function......" << endl;

    int M = b.size();
    int N = M - 2;  // Number of assets
    long double targetReturn = -b[N];

    cout << "Matrix size: " << M << "x" << M << endl;
    cout << "Number of assets: " << N << endl;

    // Step 1: Create better initial guess
    Vec x(M, 0.0L);

    // Set initial portfolio weights to equal weights that sum to 1 for guessing
    for(int i = 0; i < N; i++) {
        x[i] = 1.0L / N;
    }

    // Set initial Lagrange multipliers to reasonable values
    x[N] = targetReturn;      // λ (return constraint)
    x[N+1] = 1.0L;     // μ (budget constraint)

    cout << "Initial guess created:" << endl;
    cout << "Portfolio weight sum: " << N * (1.0L / N) << " (should be 1.0)" <<
endl;
    cout << "Initial lambda: " << x[N] << endl;
    cout << "Initial mu: " << x[N+1] << endl;

    // Step 2: Calculate initial residual r = b - Q*x
    Vec Q_x = MatM_Vec(Q, x);
    Vec r(M);
    for(int i = 0; i < M; i++) {
        r[i] = b[i] - Q_x[i];
    }
    // Step 3: Initialize search direction
    Vec p = r;  // p0 = r0

    long double tolerance = 1e-12L;  // tighter tolerance is preferred
//*********************** */
    long double rsold = dot(r, r);   // r^T * r

    cout << "Initial residual norm squared: " << rsold << endl;
    cout << "Tolerance: " << tolerance << endl;
    cout << "Start processing Conjugate Gradient Program......" << endl;

    // Step 4: Main CG iteration loop
    for (int k = 0; k < M && rsold > tolerance; ++k) {   //***********************
*/
        cout << "Iteration " << k <<
"..................................................." << endl;
        cout << "Current residual norm: " << sqrt(rsold) << endl;

        // Compute Q*p
        Vec Qp = MatM_Vec(Q, p);

        // Compute p^T * Q * p
        long double pQp = dot(p, Qp);
        cout << "p^T * Q * p = " << pQp << endl;

        // Check for stopping condition breakdown or indefinite matrix
        if (abs(pQp) < 1e-14) {
            cout << "Warning: p^T * Q * p is nearly zero (" << pQp << "), stopping"
<< endl;
            break;
        }

        // Compute step size alpha = (r^T * r) / (p^T * A * p)
        long double alpha = rsold / pQp;
        cout << "Alpha = " << alpha << endl;

        // Check for reasonable alpha
        if (abs(alpha) > 1e6) {
            cout << "Warning: Alpha is very large (" << alpha << "), may be
unstable" << endl;
        }

        // Update solution: x = x + alpha * p
        alpha_x_plus_y(alpha, p, x);

        // Update residual: r = r - alpha * A*p
        alpha_x_plus_y(-alpha, Qp, r);

        // Compute new residual norm squared
        long double rsnew = dot(r, r);
        cout << "New residual norm: " << sqrt(rsnew) << endl;

        // Check convergence
        if (sqrt(rsnew) < tolerance) {
            cout << "Converged at iteration " << k << " with residual " <<
sqrt(rsnew) << endl;
            break;
        }

        // Check for stagnation
        if (rsnew > 0.99 * rsold && k > 10) {
            cout << "Warning: Slow convergence detected at iteration " << k <<
endl;
        }

        // Compute beta = (r_new^T * r_new) / (r_old^T * r_old)
        long double beta = rsnew / rsold;
        cout << "Beta = " << beta << endl;

        // Update search direction: p = r + beta * p
        for (int i = 0; i < M; ++i) {
            p[i] = r[i] + beta * p[i];
        }

        // Update rsold for next iteration
        rsold = rsnew;

        // Safety check for maximum iterations
        if (k >= 2 * M) {
            cout << "Maximum iterations (" << 2*M << ") reached, stopping" << endl;
            break;
        }

        // Debug: Check solution progress every 10 iterations
        if (k % 10 == 0 && k > 0) {
            long double weightSum = 0.0L;
            for(int i = 0; i < N; i++) {
                weightSum += x[i];
            }
            cout << "Progress check - Current weight sum: " << weightSum << endl;
        }
    }

    cout << "Finish processing Conjugate Gradient Program......" << endl;

    // Step 5: Final verification
    Vec final_Qx = MatM_Vec(Q, x);
    long double final_residual_norm = 0.0L;
    for (int i = 0; i < M; i++) {
        long double diff = b[i] - final_Qx[i];
        final_residual_norm += diff * diff;
    }
    final_residual_norm = sqrt(final_residual_norm);

    cout << "Final residual norm: " << final_residual_norm << endl;

    // Check if any weights are NaN or Inf
    bool hasInvalidValues = false;
    for(int i = 0; i < M; i++) {
        if(isnan(x[i]) || isinf(x[i])) {
            cout << "Warning: Invalid value detected at x[" << i << "] = " << x[i]
<< endl;
            hasInvalidValues = true;
        }
    }

    if(hasInvalidValues) {
        cout << "ERROR: Solution contains NaN or Inf values!" << endl;
        // Return a reasonable fallback solution
        Vec fallback(M, 0.0L);
        for(int i = 0; i < N; i++) {
            fallback[i] = 1.0L / N;  // Equal weights
        }
        fallback[N] = 0.01L;
        fallback[N+1] = 1.0L;
        cout << "Returning fallback equal-weight solution" << endl;
        return fallback;
    }

    // Final solution summary
    long double weightSum = 0.0L;
    int negativeWeights = 0;
    for(int i = 0; i < N; i++) {
        weightSum += x[i];
        if(x[i] < 0) negativeWeights++;
    }

    cout << "\n\n\n=== SOLUTION SUMMARY ===" << endl;
    cout << "Portfolio weight sum: " << weightSum << " (should be ~1.0)" << endl;
    cout << "Number of negative weights (short positions): " << negativeWeights <<
endl;
    cout << "Lambda (return multiplier): " << x[N] << endl;
    cout << "Mu (budget multiplier): " << x[N+1] << endl;
    cout << "Final residual norm: " << final_residual_norm << "\n\n" <<endl;

    return x;
}



// Function to write Matrix2D to CSV file
void PortfolioOptimizer::writeMatrixToCSV(const Matrix2D& matrix, const string&
filename) {
    ofstream file(filename);
    if (!file.is_open()) {
        cout << "Error: Could not open file " << filename << " for writing." <<
endl;
        return;
    }

    // Set precision for floating point numbers
    file << fixed << setprecision(10);

    for (int i = 0; i < matrix.size(); ++i) {
        for (int j = 0; j < matrix[i].size(); ++j) {
            file << matrix[i][j];
            if (j < matrix[i].size() - 1) {
                file << ",";  // Add comma between values
            }
        }
        file << "\n";  // New line after each row
    }

    file.close();
    cout << "Matrix written to " << filename << endl;
}

// Function to write Vector to CSV file
void PortfolioOptimizer::writeVectorToCSV(const Vec& vector, const string&
filename) {
    ofstream file(filename);
    if (!file.is_open()) {
        cout << "Error: Could not open file " << filename << " for writing." <<
endl;
        return;
    }

    file << fixed << setprecision(10);

    for (int i = 0; i < vector.size(); ++i) {
        file << vector[i] << "\n";
    }

    file.close();
    cout << "Vector written to " << filename << endl;
}
```

backtester.h

```
#ifndef backtester_h
#define backtester_h

#include <iostream>
#include <cmath>
#include <vector>
using namespace std;
using Vec = vector <long double>;
using Matrix2D = vector<vector<long double> >;
using Cube     = vector<Matrix2D>;

struct ComprehensiveBacktestingResults {
    Matrix2D dailyReturns;        // [target_return][day] – 21×600 matrix of daily
returns
    Matrix2D windowMeanReturns;   // [target_return][period] – original 21×50
matrix
    Matrix2D windowVariances;     // [target_return][period] – original 21×50
matrix
    Matrix2D portfolioMetrics;    // [target_return][metric] – 21×9 matrix of
portfolio metrics
};

class Backtester {
private:
    // Data storage
    int totalPeriods_;
    int windowSize_;
    int outOfSampleSize_;
    int numberAssets_;
    Matrix2D returns_vec_;
    Matrix2D mean_returns_;
    Cube covariance_returns_;

public:
    // Constructor
    Backtester(int totalPeriods, int windowSize, int outOfSampleSize, int
numberAssets);

    // Main functions (public because they're used from main)
    void computeRollingParameters();
    ComprehensiveBacktestingResults createComprehensiveBacktesting(long double
riskFreeRate = 0.02L);
    void printComprehensiveSummary(const ComprehensiveBacktestingResults& results,
long double riskFreeRate = 0.02L);

    //get functions
    vector<int> getEvaluationDates();
    Vec getPortfolioWeights(int period, long double targetReturn);

    // Utility functions
    void printDataCheck();
};
#endif
```

backtester.cpp

```cpp
#include "csv.h"
#include "port_optimization.h"
#include "para_estimation.h"
#include "backtester.h"
#include <iostream>
#include <fstream>
#include <iomanip>
#include <algorithm>
#include <vector>
#include <cmath>
#include <stdio.h>
#include <fstream>
#include <stdlib.h>
#include <sstream>
#include <numeric>

using namespace std;
using Vec = vector <long double>;
using Matrix2D = vector<vector<long double> >;
using Cube     = vector<Matrix2D>;

// ******* CONSTRUCTER *******
Backtester::Backtester(int totalPeriods, int windowSize, int outOfSampleSize, int
numberAssets)
    : totalPeriods_(totalPeriods), windowSize_(windowSize),
      outOfSampleSize_(outOfSampleSize), numberAssets_(numberAssets) {

    cout << "Backtester initialized with:" << endl;
    cout << "- Total periods: " << totalPeriods_ << endl;
    cout << "- Window size: " << windowSize_ << endl;
    cout << "- Out-of-sample size: " << outOfSampleSize_ << endl;
    cout << "- Number of assets: " << numberAssets_ << endl;
}

// ******* FUNCTION 1 *******
//Load data and compute rolling means and covariance
void Backtester::computeRollingParameters() {

    // Allocate memory for return data
    long double **returnMatrix = new long double*[numberAssets_];
    for(int i = 0; i < numberAssets_; i++) {
        returnMatrix[i] = new long double[totalPeriods_];
    }

    // Read the data from file
    string fileName = "asset_returns.csv";
    ParameterEstimator::readData(returnMatrix, fileName);

    // Transform to vector object
    returns_vec_ = ParameterEstimator::transform_to_vec(returnMatrix,
numberAssets_, totalPeriods_);

    // Compute rolling means
    cout << "\nProcessing rolling means ....";
    mean_returns_ = ParameterEstimator::rollingMeans(returns_vec_);

    // Compute rolling covariance
    cout << "\nProcessing rolling covariance ....";
    covariance_returns_ = ParameterEstimator::rollingCovariance(returns_vec_,
mean_returns_);

    // Clean up memory
    for(int i = 0; i < numberAssets_; i++) {
        delete[] returnMatrix[i];
    }
    delete[] returnMatrix;

    cout << "\nRolling parameters computed successfully!" << endl;
}

// ******* FUNCTION 2 *******
//Get portfolio weights for one specific period and target return
Vec Backtester::getPortfolioWeights(int period, long double targetReturn) {

    cout << "Processing weights for period " << period << " (target " <<
targetReturn << ")..." << endl;

    // Extract covariance matrix for this period
    Matrix2D Sigma = covariance_returns_[period];

    // Extract mean returns for this period
    Vec meanReturn(numberAssets_);
    for(int i = 0; i < numberAssets_; ++i) {
        meanReturn[i] = mean_returns_[i][period];
    }

    // Build KKT matrix and RHS vector
    Matrix2D Q = PortfolioOptimizer::KKTMatrix_Q(Sigma, meanReturn);
    Vec b = PortfolioOptimizer::RHS_b(numberAssets_, targetReturn);

    // Solve for weights
    Vec x = PortfolioOptimizer::solveKKT_x(Q, b);

    // Extract just the portfolio weights (first n elements)
    Vec weights(numberAssets_);
    for(int i = 0; i < numberAssets_; ++i) {
        weights[i] = x[i];
    }


    cout << "\n\n\nWeights for period " << period << " (target " << targetReturn <<
"):\n";
    for (int i =0; i <numberAssets_;++i) {
        cout<< "Asset "<< i <<" weight:" << x[i]<<endl;
    }

    // Verify weights sum to 1
    long double sum_w = 0.0L;
    for(int i = 0; i < numberAssets_; ++i) {
        sum_w += weights[i];
    }
    cout << "Sum of weights = " << sum_w << endl;

    return weights;
}


// ******* FUNCTION 3 *******
// Print data check (utility function)
void Backtester::printDataCheck() {
    cout << "Checking:\n\n\nChecking rolling means ....";
    cout << "\nCheck mean returns - Asset 0, Period 98 : " << mean_returns_[0][98];
    cout << "\nCheck mean returns - Asset 0, Period 99 : " << mean_returns_[0][99];
    cout << "\nCheck mean returns - Asset 82, Period 699 : " <<
mean_returns_[82][699];
    cout << "\nCheck mean returns - Asset 82, Period 500 : " <<
mean_returns_[82][500] << endl;
    cout << "\nChecking rolling covariance ....";
    cout << "\nCheck covariance - Period 98, Comp0, Comp2 : " <<
covariance_returns_[98][0][2];
    cout << "\nCheck covariance - Period 99, Comp0, Comp2: " <<
covariance_returns_[99][0][2];
    cout << "\nCheck covariance - Period 99, Comp0, Comp0: " <<
covariance_returns_[99][0][0];
    cout << "\nCheck covariance - Period 99, Comp2, Comp0: " <<
covariance_returns_[99][2][0];
    cout << "\nCheck covariance - Period 99, Comp17, Comp0: " <<
covariance_returns_[99][17][0];
    cout << "\nCheck covariance - Period 99, Comp41, Comp82: " <<
covariance_returns_[99][41][82];
    cout << "\nCheck covariance - Period 99, Comp82, Comp41: " <<
covariance_returns_[99][82][41];
    cout << "\nCheck covariance - Period 99, Comp82, Comp82: " <<
covariance_returns_[99][82][82] << endl;
}


// ******* FUNCTION 4 *******
//Identify which day point I need to solve the weights.
//For example, if I want 99th period (100th day), then add 12days until 699-12=
687th period (688th day)
vector<int> Backtester::getEvaluationDates() {
    vector<int> endPeriods;
    int firstEnd = windowSize_ - 1;  // 99
    int lastEnd = totalPeriods_ - outOfSampleSize_ - 1;  // 687

    for (int end = firstEnd; end <= lastEnd; end += outOfSampleSize_) {
        endPeriods.push_back(end);
    }

    //Print Evaluating Start Dates for Each Window:
    cout << "\nEvaluation dates: ";
    for (int i = 0; i < endPeriods.size(); i++) {
        cout << endPeriods[i];
        if (i < endPeriods.size() - 1) cout << ", ";
    }
    cout << "\nTotal windows: " << endPeriods.size() << endl;

    return endPeriods;
}

/* ================================================================================
                                Column indices:
[0] = Target return, [1] = Daily return, [2] = Daily variance, [3] = Daily std dev
[4] = Annual return, [5] = Annual variance, [6] = Annual std dev
[7] = Sharpe ratio, [8] = Number of periods
=================================================================================
*/

// ******* FUNCTION 5 *******
// Expanded function that creates both window-based and daily return matrices
ComprehensiveBacktestingResults Backtester::createComprehensiveBacktesting(long
double riskFreeRate) {
    vector<int> evaluationDates = getEvaluationDates();

    // Generate target returns: 0.0%, 0.5%, 1.0%, ..., 10.0%
    vector<long double> targetReturns;
    for (int i = 0; i <= 20; i++) {
        targetReturns.push_back(i * 0.005);
    }

    cout << "Creating comprehensive backtesting results..." << endl;
    cout << "Target returns: " << targetReturns.size() << endl;
    cout << "Evaluation dates: " << evaluationDates.size() << endl;
    cout << "Days per window: " << outOfSampleSize_ << endl;

    // Calculate total days (50 windows × 12 days = 600 days)
    int totalDays = evaluationDates.size() * outOfSampleSize_;
    cout << "Total out-of-sample days: " << totalDays << endl;

    //CONSTRUCT DATA TYPE results (include all 4 datatypes)
    ComprehensiveBacktestingResults results;

    // INITIALIze matrices according to "struct" (smart way to declare with struct)
    results.dailyReturns = Matrix2D(targetReturns.size(), Vec(totalDays, 0.0L));
    results.windowMeanReturns = Matrix2D(targetReturns.size(),
Vec(evaluationDates.size(), 0.0L));
    results.windowVariances = Matrix2D(targetReturns.size(),
Vec(evaluationDates.size(), 0.0L));
    results.portfolioMetrics.resize(targetReturns.size(), Vec(9, 0.0L));  // 21×9
matrix

    // Loop through each target return - t means target yearly return
    for (int t = 0; t < targetReturns.size(); t++) {
        long double targetReturn = targetReturns[t];

        cout << "\nProcessing target return " << (t+1) << "/" <<
targetReturns.size()
```

```
             << ": " << targetReturn*100 << "%" << endl;

        vector<long double> allDailyReturns;  // Collect all 600 daily returns for
this target
        allDailyReturns.reserve(totalDays); //reserve memory space for push_back,
so here reserve 600 days

        int dayIndex = 0;  // Index for daily returns matrix

        // WINDOW BY WINDOW
        // Loop through each evaluation period (50 windows)
        for (int p = 0; p < evaluationDates.size(); p++) {
            int period = evaluationDates[p];

            if (p < 5 || p % 10 == 0) {
                cout << "  Window " << (p+1) << "/" << evaluationDates.size()
                     << " (rebalance at day " << period+1 << ")" << endl;
            }

            // Get NEW portfolio weights for this window (rebalancing every 12
days!)
            Vec weights = Backtester::getPortfolioWeights(period, targetReturn);

            // Calculate out-of-sample returns using these weights for next 12 days
            Vec windowReturns; //create a vector of 12 days (in next line) --> re-
declare in each window
            windowReturns.reserve(outOfSampleSize_);

            for (int day = 1; day <= outOfSampleSize_; day++) {
                int returnPeriod = period + day;

                // Calculate portfolio return for this day
                long double dailyReturn = 0.0L;
                for (int asset = 0; asset < numberAssets_; asset++) {
                    dailyReturn += weights[asset] *
returns_vec_[asset][returnPeriod];
                }

                // Store in both matrices
                results.dailyReturns[t][dayIndex] = dailyReturn;  // Daily returns
matrix
                windowReturns.push_back(dailyReturn); // this is declared within
the for loop
                allDailyReturns.push_back(dailyReturn); //this is declared earlier
already

                dayIndex++;
            }

            // ********** Calculate WINDOW STATISTICS (mean and variance for this
12-day period) **********
            long double windowSum = 0.0L;
            for (long double ret : windowReturns) {
                windowSum += ret;
            }
            long double windowMean = windowSum / windowReturns.size();

            long double windowVariance = 0.0L;
            for (long double ret : windowReturns) {
                long double diff = ret - windowMean;
                windowVariance += diff * diff;
            }
            if (windowReturns.size() > 1) {
                windowVariance /= (windowReturns.size() - 1);
            }

            // Store window statistics
            results.windowMeanReturns[t][p] = windowMean;
            results.windowVariances[t][p] = windowVariance;
        }
        // ********** ABOVE already calculated WINDOW STATISTICS (mean and variance
for this 12-day period) **********


        // ********** Start calculating ALL-TIME STATISTICS (mean and variance for
this 12-day period) **********
        // Calculate comprehensive portfolio metrics using ALL 600 daily returns
        int numPeriods = allDailyReturns.size(); //600 DAYS

        // Calculate overall mean return
        long double totalSum = 0.0L;
        for (long double ret : allDailyReturns) {
            totalSum += ret;
        }
        long double dailyMeanReturn = totalSum / allDailyReturns.size();

        // Calculate overall variance (true 600-day variance with rebalancing)
        long double variance = 0.0L;
        for (long double ret : allDailyReturns) {
            long double diff = ret - dailyMeanReturn;
            variance += diff * diff;
        }
        if (allDailyReturns.size() > 1) {
            variance /= (allDailyReturns.size() - 1);
        }
        long double totalStdDev = sqrt(variance);

        // Annualize metrics (252 trading days per year)
        long double annualizedReturn = dailyMeanReturn * 252.0L;
        long double annualizedVariance = variance * 252.0L;
        long double annualizedStdDev = sqrt(annualizedVariance);

        // Calculate Sharpe ratio
        long double sharpeRatio = 0.0L;
        if (annualizedStdDev > 1e-8) {
            sharpeRatio = (annualizedReturn - riskFreeRate) / annualizedStdDev;
        }

        // Store comprehensive portfolio metrics in Matrix2D format
        results.portfolioMetrics[t][0] = targetReturn;           // Target return
        results.portfolioMetrics[t][1] = dailyMeanReturn;           // Daily mean
return
        results.portfolioMetrics[t][2] = variance;              // Daily variance
        results.portfolioMetrics[t][3] = totalStdDev;           // Daily std dev
        results.portfolioMetrics[t][4] = annualizedReturn;      // Annual return
        results.portfolioMetrics[t][5] = annualizedVariance;    // Annual variance
        results.portfolioMetrics[t][6] = annualizedStdDev;      // Annual std dev
        results.portfolioMetrics[t][7] = sharpeRatio;           // Sharpe ratio
        results.portfolioMetrics[t][8] = (long double)numPeriods; // Number of
periods

        cout << "  → Daily return: " << fixed << setprecision(4) <<
dailyMeanReturn*100
             << "%, Annual return: " << annualizedReturn*100
             << "%, Sharpe: " << sharpeRatio << endl;
    }

    cout << "\n=== COMPREHENSIVE BACKTESTING COMPLETED ===" << endl;
    cout << "Daily returns matrix: " << results.dailyReturns.size() << " × " <<
results.dailyReturns[0].size() << endl;
    cout << "Window statistics: " << results.windowMeanReturns.size() << " × " <<
results.windowMeanReturns[0].size() << endl;

    //  =========== Export all CSV files ===========
    cout << "\nExporting results to CSV files..." << endl;

    // 1. 21 × 50 window return statistics
    PortfolioOptimizer::writeMatrixToCSV(results.windowMeanReturns,
"21x50_window_returns.csv");
    cout << "Exported: 21x50_window_returns.csv" << endl;

    // 2. 21 × 50 window variance statistics
    PortfolioOptimizer::writeMatrixToCSV(results.windowVariances,
"21x50_window_variances.csv");
    cout << "Exported: 21x50_window_variances.csv" << endl;

    // 3. 21 × 600 daily return statistics
    PortfolioOptimizer::writeMatrixToCSV(results.dailyReturns,
"21x600_daily_returns.csv");
    cout << "Exported: 21x600_daily_returns.csv" << endl;

    // 4. 21 × N portfolio metrics (already in matrix format)
    PortfolioOptimizer::writeMatrixToCSV(results.portfolioMetrics,
"21xN_portfolio_metrics.csv");
    cout << "Exported: 21xN_portfolio_metrics.csv" << endl;

    // Create header file for portfolio metrics CSV
    ofstream headerFile("portfolio_metrics_header.txt");
    if (headerFile.is_open()) {
        headerFile << "Column headers for 21xN_portfolio_metrics.csv:" << endl;
        headerFile << "Column 0: Target Return (decimal)" << endl;
        headerFile << "Column 1: Daily Mean Return (decimal)" << endl;
        headerFile << "Column 2: Daily Variance" << endl;
        headerFile << "Column 3: Daily Standard Deviation" << endl;
        headerFile << "Column 4: Annualized Return (decimal)" << endl;
```

```cpp
        headerFile << "Column 5: Annualized Variance" << endl;
        headerFile << "Column 6: Annualized Standard Deviation" << endl;
        headerFile << "Column 7: Sharpe Ratio" << endl;
        headerFile << "Column 8: Number of Periods" << endl;
        headerFile.close();
        cout << "✓ Exported: portfolio_metrics_header.txt" << endl;
    }

    cout << "\nAll CSV files exported successfully!" << endl;
    cout << "File Summary for User:" << endl;
    cout << "- 21x50_window_returns.csv: Mean returns for each window" << endl;
    cout << "- 21x50_window_variances.csv: Variances for each window" << endl;
    cout << "- 21x600_daily_returns.csv: Daily portfolio returns with rebalancing"
<< endl;
    cout << "- 21xN_portfolio_metrics.csv: Comprehensive portfolio statistics" <<
endl;
    cout << "- portfolio_metrics_header.txt: Column descriptions for metrics file"
<< endl;

    return results;
}

// ******* FUNCTION 6 *******
// Function to print comprehensive summary for reference
void Backtester::printComprehensiveSummary(const ComprehensiveBacktestingResults&
results,
                              long double riskFreeRate) {

    cout << "\n" << string(100, '=') << endl;
    cout << "COMPREHENSIVE PORTFOLIO PERFORMANCE SUMMARY" << endl;
    cout << string(100, '=') << endl;

    cout << "Risk-free rate: " << riskFreeRate*100 << "%" << endl;
    cout << "Rebalancing: Every 12 days (50 rebalancing periods)" << endl;
    cout << "Analysis period: 600 out-of-sample days" << endl;
    cout << endl;

    cout << setw(8) << "Target%" << setw(12) << "Daily%" << setw(12) << "Annual%"
         << setw(12) << "Ann.StdDev%" << setw(12) << "Sharpe" << setw(12) << "Days"
<< endl;
    cout << string(100, '-') << endl;

    for (int t = 0; t < results.portfolioMetrics.size(); t++) {
        long double targetReturn = t * 0.005;

        cout << fixed << setprecision(1);
        cout << setw(8) << targetReturn * 100;
        cout << setprecision(3);
        cout << setw(12) << results.portfolioMetrics[t][1] * 100;  // Daily return
        cout << setprecision(2);
        cout << setw(12) << results.portfolioMetrics[t][4] * 100;  // Annual return
        cout << setw(12) << results.portfolioMetrics[t][6] * 100;  // Annual std
dev
        cout << setprecision(3);
        cout << setw(12) << results.portfolioMetrics[t][7];        // Sharpe ratio
        cout << setw(12) << (int)results.portfolioMetrics[t][8];   // Num periods
        cout << endl;
    }

    cout << string(100, '-') << endl;

    // Find best performing portfolios
    int bestSharpeIndex = 0, bestReturnIndex = 0;
    long double maxSharpe = results.portfolioMetrics[0][7];
    long double maxReturn = results.portfolioMetrics[0][4];

    for (int t = 1; t < results.portfolioMetrics.size(); t++) {
        if (results.portfolioMetrics[t][7] > maxSharpe) {
            maxSharpe = results.portfolioMetrics[t][7];
            bestSharpeIndex = t;
        }
        if (results.portfolioMetrics[t][4] > maxReturn) {
            maxReturn = results.portfolioMetrics[t][4];
            bestReturnIndex = t;
        }
    }

    cout << "\nKEY FINDINGS:" << endl;
    cout << "Best Sharpe Ratio: " << setprecision(3) << maxSharpe
         << " (Target: " << bestSharpeIndex*0.5 << "%)" << endl;
    cout << "Highest Return: " << setprecision(2) << maxReturn*100
         << "% (Target: " << bestReturnIndex*0.5 << "%)" << endl;

    cout << string(100, '=') << endl;
}
```

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

```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];
}
```